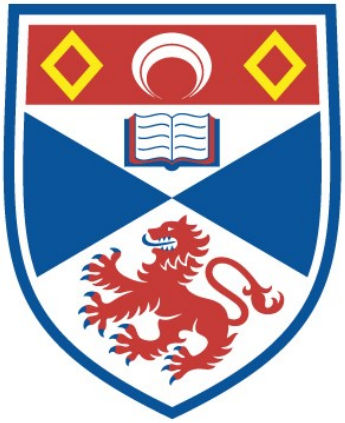
VERIFIED PROGRAMMING WITH EXPLICIT COERCIONS

Christopher Schwaab

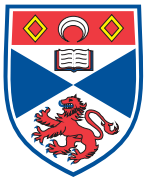
A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2019

Verified programming with explicit coercions

Christopher Schwaab



University
of
St Andrews

This thesis is submitted in partial fulfillment for the degree of
Doctor of Philosophy (PhD)
at the University of St Andrews

March 2018

Abstract

Type systems have proved to be a powerful means of specifying and proving important program invariants. In dependently typed programming languages types can depend on values and hence express arbitrarily complicated propositions and their machine checkable proofs. The type-based approach to program specification allows for the programmer to not only transcribe their intentions, but arranges for their direct involvement in the proving process, thus aiding the machine in its attempt to satisfy difficult obligations. In this thesis we develop a series of patterns for programming in a correct-by-construction style making use of constraints and coercions to prove properties within a dependently typed host. This allows for the development of a verified, kernel which can be built upon using the host system features. In particular this should allow for the development of "tactics" or semi-automated solvers invoked when coercing types all within a single language. The efficacy of this approach is given by the development of a system of expressions indexed by their, exposing a case analysis feature serving to generate value constraints. These constraints are directly reflected into the host allowing for their involvement in the type-checking process. A motivating use case of this design shows how a term's semantic index information admits an exact, formalized cost analysis amenable to reasoning within the host. Finally we show how such a system is used to identify unreachable dead-code, trivially admitting the design and verification of an SSA style compiler with this optimization. We think such a design of explicitly proving the local correctness of type-transformations in the presence of accumulated constraints can form the basis of a flexible language in concert with a variety of trusted solver.

Declaration

Candidate's declaration

I, Christopher Joseph Schwaab, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 65,000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in September 2013.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Date Signature of candidate

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date Signature of supervisor

Permission for publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Christopher Joseph Schwaab, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

Printed copy

No embargo on print copy.

Electronic copy

No embargo on electronic copy.

Date Signature of candidate

Date Signature of supervisor

Unerpinning Research Data or Digital Outputs

Candidate's declaration

I, Christopher Joseph Schwaab, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

Date Signature of candidate

Acknowledgements

This work was supported by the University of St Andrews (School of Computer Science); as a part of the University of St Andrews 600^{th} anniversary scholarship.

Becky, thank you. This work would have never been completed without your support and love. Esme, thank you for always bringing a smile to me throughout these years of work and Oran, thank you for not arriving on the day of my viva.

My great thanks to Professor Jeremy Siek, without your help, patience, and guidance I would never have embarked on this journey. Thank you for your continued support throughout my PhD during your visits to Edinburgh.

I would also like to thank the research group and in particular those who I shared an office with at St Andrews. Matúš, Adam, Franck, David, speaking with you was always illuminating and your suggestions have been invaluable. Thank you also to my supervisors Edwin Brady and Kevin Hammond for your help during these years.

Thanks to Tom Schrijvers for all the time you spent teaching me and working with me. I found my time in Ghent very valuable.

Finally, thank you to the wider Scottish programming language community. I greatly enjoyed the time spent sharing ideas and excitement. In particular thank you to Conor McBride, I always appreciated your encour-

agement.

Contents

Chapter 1

Introduction

Writing correct software is a difficult task. With the increasing ubiquity of computing devices in our lives and in our pockets, the importance of ensuring the safe and correct evaluation of programs is only going to grow. These bugs range from incorrect problem specifications, incorrect transcriptions of correct specifications, to improper respect for language semantics. In an attempt to stymie these errors a variety of both "offline" and "online" strategies exist. "Offline" or static analysis concerns itself with the checking of program source-code, intermediate forms, and compiled program forms without the need to execute the code itself. "Online" or dynamic analysis performs correctness checking as a part of program execution. It is also possible to combine aspects of both static and dynamic checking into hybrid strategies, usually involving the dynamic checking of special test code written alongside the program which is not intended to be compiled into the main program.

Runtime safety with dynamic checking

Dynamic checking of programs is often built into the runtime system of the language used for development. For example languages such as Python, Ruby, and Erlang [AVW93] are well known. Arithmetic operations and indexing can be checked as they are performed, ensuring problems of overflow and out of bounds accesses are suitably handled. Additionally, the *type* of the data encoded by a variable can be tracked and operations suitably restricted. For instance, it is unlikely that a string and an integer can be reasonably added together. Such checks primarily help protect the programmer from

incorrectly implementing their specifications.

Compile time safety with static checking

Alternatively, errors can be statically checked and indeed primitive type classification of values in compilers date back to ALGOL [Hab71]. However statically checking *properties* of computations such as arithmetic and indexing operations is considerably more difficult and are often implemented in external tools. Some examples are model-checking [DKW08] whereby correctness properties are specified as logical formulae and often automatically checked by an internal prover. Another popular form of static checking is *abstract interpretation* [CC77], used to check specific properties such as the validity of pointer dereferences in C by evaluating some subset of the program in an abstract domain, thus avoiding the need to model side-effects such as cache, program input output (I/O), etc.

Due to this high difficulty of correcntess checking, many statically typed languages take a hybrid approach. For example, C# performs type-checking at compile time but overflow and bounds checking at runtime.

Hybrid approaches also exist outside the language runtime, requiring some amount of setup by the program develop. Unit tests are a prominant example whereby small "units" of a program or library are specifically tested. A programmer first implements a test and then compiles and runs it, checking some expected outputs. These units can be anything from a single function, to a class in an object-oriented language, or even a module which for example might consist of several objects and services working in orchestration. In unit testing a developer usually defines some invariants or correctness properties and then tests these properties hold, or error conditions are handled, given a variety of programs states and inputs. Some unit testing suites additionally provide a means of automatically generating inputs. Generating somehow "unexpected" inputs and ensuring these are properly handled by the program is often referred to as "fuzzing".

While tools exist to perform static control and data flow analysis to discern a program's safety, the checks are often incomplete due to the undecidability of the analyses. One simple solution is to push this problem onto the developer by providing a system for formally specifying not only a program's expected properties, but additionally the proofs that these properties hold. Isabelle [Wen+17b] and Coq [Tea18] are two prominant examples of such *formal verification* systems. Due to the time intensity of

manual verification, lightweight checking can be done e.g. by programming in pairs, or by requiring developers to "check-off" some correctness properties as a part of a "code-review" a session in which non-authors of a program look over and comment on the correctness of a piece of code.

As another alternative, program properties can be embedded in types augmented to inform not only a value's primary classification but additionally their distinguished features such as boundedness. Thus types allow the programmer to arrange and make locally available, any evidence required to prove the correctness of an operation. Since such evidence is itself a value whose type *depends* upon the values it restricts, these types are called dependent types.

Dependent types

Dependent types allow the programmer to precisely provide the machine with a specification for a program's operation. Thus the compiler can automatically check a program satisfies the specification, statically eliminating any number of difficult to find bugs and unusual program states. The availability of such highly specific type information not only allows for safety properties to be checked but additionally exposes optimization opportunities to the compiler. For instance Xi [Xi99]; [XP98] successfully used type information to eliminate bounds checks and improve dead code elimination in Dependent ML. However, this comes at the expense of heavily annotated types [RKJ08]. This strength of generality in the problems that can be solved is also a weakness since the system cannot be tailored to any one particular problem domain. Thus as an alternative to the use of a general theorem prover, Domain Specific Languages (DSL) can be used.

Domain specific languages

Such languages are highly specific and often best suited to solve a single problem domain; however, this allows the programming paradigm to be tailored to an area of interest and thus simplifying the task of development. DSL's have become popular in recent years, for example, R [IG96] allows for the rapid development and prototyping of statistical analyses. The database BigQuery [Sat12] can operate on datasets that would take days in another language, including implementations of the SQL DSL such as MySQL. Theorem provers can also make use of DSLs. The theorem

prover Coq exposes a DSL $\mathcal{L}_{tac}$ [Del00] for giving proofs by repeated goal refinement by the application of "tactics." DSLs are even used at the kernel level, for example, Linux makes use of the Berkeley Packet Filter (BPF) architecture [MJ93] and more recently the nftables project for user-level packet processing. Linux has even recently gained a Just-in-Time compiler (JIT) for compiling BPF code on the fly. Another advantage of the restricted problem domain in which DSLs operate is the ease on the burden of proving program correctness. Since the scope of allowable terms is more restricted, the problem of ruling out some "wrong" behavior is simplified.

Exemplifying the need for safety guarantees is the language DAML [Ass16] of DigitalAssets, a DSL for the development of "smart contracts" seeking to "ensure legal certainty, distributed execution, privacy, ease of analysis and ease of use." This is particularly important given its intended use in markets with well established legal precedent. In more general languages for smart contract development such as Solidity [mai18] without any guarantees, bugs can be incredibly costly. For instance, a bug in the DAO contract cost investors over 50 million USD and famously resulted in the forking of the ethereum blockchain [Woo18] into ethereum and ethereum classic. The use of a restricted DSL is particularly crucial in the area of markets where strict legal obligations must be fulfilled.

While DSLs present a powerful means of solving problems their development is often an intensive task, requiring the construction of a full compiler stack from the tokenization and parsing of string to optimized machine code. As a lightweight alternative to such a task, embedded domain-specific languages (EDSL) can be developed.

Embedded domain specific languages

The "embedded" term indicates that the DSL is embedded within some existing language. By leveraging an existing language EDSL development not only avoid the need for a parser, but can additionally rely on the host for type-checking, optimization, and even object code emission although this is not strictly necessary. Ivory [Ell+15], a language for real-time critical systems is implemented within Haskell and makes extensive use of Haskell's sophisticated kind system. Ivory gives strong guarantees of type and memory safety, using kinds to statically reason about memory regions and object layout within memory. Ivory not only allows for compilation to machine code but can also emit user declared propositions for verification by external

model checkers. The development of Ivory as a DSL has allowed for a careful crafting of paradigms supporting ease of worst case execution time analysis (WCET), stack consumption analysis, and pointer validity guarantees. In particular by taking suitably restricted expressions and looping constructs, the existence of statically verifiable bounds can be guaranteed, allowing for both termination checking and precise cost analysis. In this thesis, a similar strategy is employed.

Feldspar [Axe+10] is another EDSL in Haskell which first allows Haskell to heavily optimize its code before performing further optimization and code emission. Such a "deep-shallow" strategy allows the host to perform inlining of "macros" that construct Feldspar syntax trees to be consumed by its compiler. Since the compiler can only operate over syntax trees, these high-level macros are guaranteed to have been evaluated by the code emission stage. This strategy allows for the development of "fusion" laws which merge some classes of operations such as repeated array traversals. The works developed in this thesis make extensive use of feldspar style macros eliminating the need for direct function support.

EDSLs have been successfully used to perform everything from high-level tasks such the CoddFish language supporting database programming [SV06], all the way down to the level of hardware development via embedded hardware description languages such as Lava and CλaSH [Gil+09]; [Baa+10]. The use of EDSLs not only support the compiler's ability to emit performant code but additionally support the programmer with a natural API directly suited to the task at hand. This supports both the compiler's ability to perform automated correctness checking and the developer's ability to reason about the implementation and program state.

1.1 EDSL design with dependent types

Functional languages have long enjoyed natural support for the development of EDSLs. Their strong type-systems relative to mainstream imperative languages such as Java—initially lacking basic type-safety [Sar97]—simplify safety properties whereby the programmer's mere construction of a term guarantees its correctness. This "correct-by-construction" style of theorem proving allows for developers without any formal training in mechanized theorem proving to work with the machine to prove their code's soundness. Dependently typed languages are thus particularly well suited to EDSL

construction courtesy of their highly expressive type-system, allowing for the embedding of sophisticated predicates over terms in the embedded language. Further, the use of *reflection*—a paradigm unique to dependent types—enables information captured by EDSL syntax to be pushed directly into the type system and reasoned about.

This thesis focuses on exploring the extent to which the affordances offered by modern dependently typed languages can be leveraged in order to prove correctness properties about some classes of programs. To this end, a prototype language $\mathscr{L}$ is defined within Idris. Programs in $\mathscr{L}$ are indexed by their semantics, an auxiliary language of index expression (often written as Ix or Semantics), ensuring that any index respecting transformation is semantically correct by construction. A key feature of $\mathscr{L}$ is that type-coercions are written explicitly, allowing for an experiment in specifying coercions *semantically* using a value's index. Finally, we explore how the host's evaluation facilities can be leveraged to aid in developing $\mathscr{L}$ programs, and performing semi-automated theorem proving.

While this platform is ideal for exploring the extent to which dependent types simplify this style of correct by construction programming, exploring how to use these ideas in a DSL with some well-defined domain is an area of future work.

1.2 Overview

Motivating the design and feature set of $\mathscr{L}$ and its analysis is an implementation of a full compilation stack, in addition to machine checked correctness proofs where viable. Figure 1.1 gives a high-level overview of the stages in compilation where dashed lines indicate unimplemented but feasible paths.

At the top of the diagram is $\mathscr{L}$, the language of expressions that is directly programmed in. Expressions are strongly typed using the type system found in the simply typed λ-calculus. However, each term also carries an "index expression," additional type information capturing the term's semantics. The expression language $\mathscr{L}$ lacks any primitive operations, such as arithmetic functions, and thus is not suitable for direct compilation or analysis as it is too high-level.

Programs written in $\mathscr{L}$ undergo a type and index expression preserving transformation to a statement language $\mathscr{S}$. This transformation decomposes the inefficient peano number representation, and low-level addition

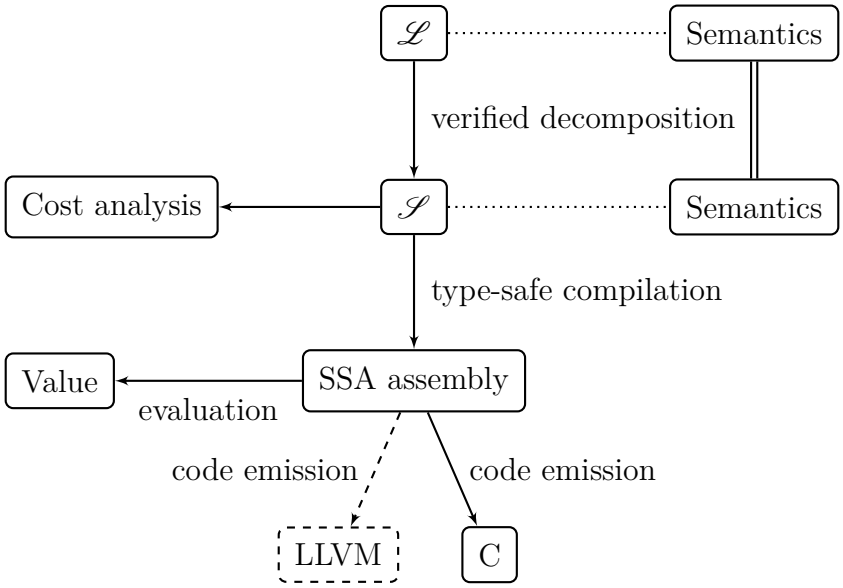


Figure 1.1: System design and compilation scheme.

operation on numbers into high-level operations suitable for compilation and analysis. The correctness of this pass seeks to show the viability of correct-by-construction proofs given by indexing terms by their semantics.

After high-level operations have been extracted, the syntax tree is ideal for static analysis or the direct compilation to an abstract machine. The analysis given is a static cost analysis using symbolic evaluation. Alternatively, the syntax tree can be compiled to machine instructions using a traditional continuation passing scheme. This scope safe, type-preserving compilation pass has been mechanized in a correct-by-construction style within Idris.

Finally from a program's abstract machine code an executable file can be produced or directly evaluated under the semantics of the abstract machine. Since the machine's instruction set is in SSA style LLVM code can be readily emitted, however for simplicity and proof-of-concept we have decided to emit plain C. The emitted C code must additionally link against a small RTS implementing the machine's intrinsics for list allocating and manipulation.

1.3 Contributions

- Explore the engineering challenges of language development with dependent types.
- Develop and apply a collection of programming patterns for use with semantics indexed terms in a dependently typed language.
- Develop dependently typed programming patterns for working with verified type coercions.
- Give a proof of concept language $\mathscr{L}$, along with a compiler down to an assembly style virtual machine using the semantic index respecting, correct-by-construction style.
- Show how to integrate a generic monoid solver "tactic" developed within the host language Idris. This solver is used to aid in performing explicit type coercions in $\mathscr{L}$.

1.3.1 Chapter Outline

The three primary levels in figure 1.1 are the focus of this thesis, and partitioned by chapters. The structure of chapter is as follows

Chapter 2 explores the background of successively more powerful type systems culminating in the three primary forms of dependent types. Additionally a basic introduction to programming and proving with full-spectrum dependent types in Idris is given. Finally a small collection of programming patterns and theorem proving strategies using dependent types are reviewed, including proof by reflection.

Chapter 3 motivates and develops the language of index expression and $\mathscr{L}$ the language of expressions indexed by their semantics (given as an index expression). The chapter iteratively develops a means of performing safe, explicit coercions using a novel constraint context reflection mechanism. Then a powerful means of mechanically performing static reasoning about expressions via their far simpler index expressions is developed, along with a fully automated, mechanized proof of transporting correctness properties from index expressions to expressions. Finally the language is extended with support for proving

or disproving the reachability of branches in case statements, allowing for safe dead-code elimination in a correct-by-construction style.

Chapter 4 argues for the feasibility of the approach by exploring the development of basic functions and how to prove properties about them. Additionally a pattern for programming and proving with macros in the "deep-shallow" style of Feldspar is developed.

Chapter 5 shows how index expressions can be used to verify the correctness of structural transformations on syntax trees. Next a cost analysis on $\mathscr{S}$ is developed showing how the normalizer of the host language Idris can automatically provide an EDSL with a symbolic evaluator. The cost analysis is directed by the statements' index expressions. Finally a continuation passing compiler which is both scope and type-safe is developed.

Chapter 6 presents some conclusions in addition to exploring areas of future work, limitations of the reference implementation along which potential strategies for improvements.

Chapter 2

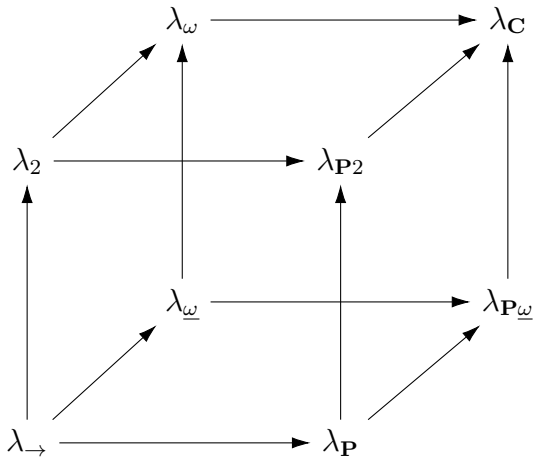
Programming with Dependent Types

One of the significant advantages of implementing an embedded domain-specific language rather than a concrete DSL is, for example, to take advantage of the host's compiler optimization as is done by meta-repa [AS13] and the host's type-system. For example, the language Ivory [Ell+15] embedded in Haskell as a safe systems programming language leverages the type-system to perform checking of memory safety properties and admit the possibility of statically checked assertions. Embedded domain-specific languages also come in a much simpler form on a smaller scale, providing a specific API targeting a highly-specific task rather than a language to be compiled. For example, the Selda Haskell library performs compile-time checking of SQL operations but is meant to be used in existing Haskell programs.

We will develop a language $\mathscr{L}$ that curiously has no syntactic means of discerning term equality, instead realying on the host system similar to the development of Kipling by McBride [McB10b]. It will be shown that the primary idea required is *convertibility* via semantic equality. By specifying convertibility in terms of features of the host, all the existing host features can be used during coercion of a term's type. The correctness of this style of implementation is readily proved correct. A brief overview of the existing work on dependent types will be covered.

2.1 What are dependent types?

Type systems are a means of decorating program terms or sentences with information about their intended use or semantics. In his book, *Practical Foundations for Programming Languages* Robert Harper describes a type-system as a means "to impose constraints on the formulation of phrases that are sensitive to the context in which they are used" [Har12]. Today type-systems are used for simple tasks such as stratifying values by bit widths as in C and scale up to general theorem proving as in Coq and Idris. The features of type-systems found in many modern-day functional languages and theorem provers can be broadly broken down into three coarse typing structures: System F, System F_ω, and dependent types. Barendregt identified these three systems along with five others in his work *Introduction to generalized type systems* [Bar91] identifying the "fine-structure" of the systems where moving between any two corresponds to adding or removing a feature. Barendregt visualized these systems into the "lambda cube"



Each corner corresponds to a type system, and each arrow corresponds to augmenting the source type-system with some quantification, or allowing for some new form of type/value dependency.

Languages on the top face of the cube have values which can depend on types. This corresponds to a system with polymorphic functions.

Languages on the back face of the cube have types which can depend on types. This corresponds to allowing for type-level functions, a feature

enjoying only limited support in the most popular programming languages of today. The best example of such a programming language is GHC's version of Haskell which has a *type family* feature, allowing for direct type-level functions.

Languages on the right side of the cube allow for types to depend on values e.g. as supported by the Twelf [PS99] programming language. This of course is the primary feature supported by dependently typed programming languages such as Idris as used in this thesis.

While the exact type-theory underlying Idris itself does not appear on the cube due to its great wealth of—sometimes ad-hoc—programming feature, the top-right corner $\lambda_{\mathbf{C}}$ captures the heart of the type system: *full-spectrum dependent types*. This corresponds to a system in which types and terms can be freely mixed. Idris supports a great deal of more features however, such as type-directed function overloading (i.e. ad-hoc polymorphism), inductive families, type-classes, and more.

Of primary interest are the simply typed λ-calculus $\lambda_{\rightarrow}$, System F $\lambda_{\mathbf{2}}$, System F_ω λF_ω, and the calculus of constructions $\lambda_{\mathbf{C}}$ which are now briefly described.

The simply typed λ-calculus

Early work most closely aligned to what is called type-theory today was carried out by Church [Chu41] and Curry [CF67] in the 1940's and 1950's. Church sought out a means of freeing the foundations of mathematics from set-theory, developing the λ-calculus and the higher-order logic (HOL) [Chu40] which forms the basis of the Isabelle theorem prover [Wen+17b]. The simply typed λ-calculus extends the raw untyped λ-calculus with some set of ground types, in addition to constants and a set of primitive functions. The syntax and typing rules for the simply typed λ-calculus with natural numbers and additions is given in figure 2.1. In a highly non-standard departure from the usual notation, context extension is written on the left as in $(x, \tau), \Gamma$ rather than $\Gamma, (\tau, x)$. This notation mirrors the encoding in Idris where contexts are represented using lists, and lists are extended on the left as in `(x, tau) :: Gamma`. Additionally values x in the context are explicitly typed using a pair (x, τ) rather than the more usual notation $x : \tau$, again to mirror the development in Idris where values are typed in the context, with an explicit pair representation. Finally, the development in Idris includes natural numbers as a primitive type and thus the theories described here also

Types $\tau ::= Nat \mid \tau \to \tau$

Expressions $e ::= n \in \mathbb{N} \mid x \mid e + e \mid \lambda x.e \mid e\ e$

$\boxed{\Gamma \vdash e : \tau}$ (well-formed expressions)

$$\text{WfLit}\frac{}{\Gamma \vdash n : Nat}$$

$$\text{WfVar}\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{WfAdd}\frac{\Gamma \vdash e_1 : Nat \quad \Gamma \vdash e_2 : Nat}{\Gamma \vdash e_1 + e_2 : Nat}$$

$$\text{WfLam}\frac{(x, \tau_1), \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \text{WfApp}\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Figure 2.1: The syntax and semantics of the simply typed λ-calculus.

include natural numbers following later developments. Given its simplicity the simply typed λ-calculus is surprisingly expressive; however it notably allows only for quantification over *values*, and not over types.

System F

In the 1970's and 1980's one of the earliest languages with a "modern" type system was independently developed by Jean-Yves Girard [Gir72] and John Reynolds[Rey74]: System F, or the polymorphic λ-calculus. System F corresponds to moving up along the lambda cube, extending the simply typed λ-calculus by allowing for quantification over types and formalizing the notion of parametric polymorphism. Reynolds sought to develop a means of allowing safe, checked languages to define valid programs whose type-correctness does not rely on the particular encoding of its primitive values. In his seminal work, Reynolds gives the example of a sorting function made parametric in the underlying list value type by abstracting over a comparison function. Girard was interested in proof theory, studying constructive proofs, and a means of expressing second-order theorems [Wad01]. The syntax and typing rules of System F are presented in figure 2.2 with α ranging over type variables. One of the brilliant insights of System F is to not only introduce type-variable quantification via $\forall\alpha\ldots$ but to extend the term language of the simply typed λ-calculus with a type binder Λ. The secondary binder Λ introduces a new type variable at the term level which can only take part

Types $\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\tau$

$\boxed{\Delta \vdash \tau}$ (well-formed types)

$$\text{WfTyVar}\frac{\alpha \in \Delta}{\Delta \vdash \alpha} \qquad \text{WfAll}\frac{\alpha, \Delta \vdash \tau}{\Delta \vdash \forall \alpha.\tau}$$

$$\text{WfArrow}\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}$$

Expressions $e ::= x \mid \lambda x.e \mid \Lambda \alpha.e \mid e\ e \mid e[\tau]$

$\boxed{\Delta; \Gamma \vdash e : \tau}$ (well-formed expression)

$$\text{WfVar}\frac{(x, \tau) \in \Gamma}{\Delta; \Gamma \vdash x : \tau}$$

$$\text{WfLam}\frac{\Delta; (x, \tau_1), \Gamma \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \text{WfApp}\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{WfLamTy}\frac{\alpha, \Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda \alpha.e : \forall \alpha.\tau} \qquad \text{WfAppTy}\frac{\Delta \vdash \tau \quad \Delta; \Gamma \vdash e : \forall \alpha.\tau}{\Delta; \Gamma \vdash e[\tau'] : \tau[\alpha \mapsto \tau']}$$

Figure 2.2: System F syntax and typing rules.

in type-level operations, allowing for its safe erasure at compile time. This partitioning is enforced by a clever stratification of terms and types, where a type can only be placed in a term at type *instantiation* sites, a secondary form of application $e[\tau]$.

As an example the polymorphic identify function can be written as

$$id : \forall \alpha.\alpha \to \alpha$$
$$id = \Lambda \alpha.\lambda x.x$$

then to apply this function, it must be instantiated at a concrete type. Supposing some ground type *Nat* were introduced, the following could be written

$$fortyTwo : Nat$$
$$fortyTwo = id[Nat]\ 42$$

This notion of stratifying classes of objects into two *phases*—those statically evaluated and those which must be delayed until runtime—using distinct binders has been used across many areas since it readily gives a guarantee that some quantified value cannot take part in runtime computations.

Remarkably the type-level quantification of System F induces a *parametricity result* discovered by Wadler [Wad89] which by interpreting types as propositions allows for the mechanical derivation of theorems about polymorphic functions. The idea is founded on the restrictions placed upon a function over some abstract type a. Since the function knows nothing about values in a, inspection and direct manipulation are not possible; thus the function is restricted to shunting the value through existing structures.

The polymorphic λ-calculus adds considerable expressive power to the simply typed λ-calculus; however, types in this system are *single kinded*. The type of a type is often called the *kind* of that type e.g. the kind of *Nat* is $\star$. The natural extension to System F expands the available kinds.

System F_ω

Moving along the lambda cube to the back face to λ_ω yields System F_ω, extending System F with type-level abstractions. Thus System F_ω has at least two kinds, the kind $\star$ of "vanilla" types, and the kind $\kappa_1 \to \kappa_2$ of type-level functions, extending the power of System F with support for type-level computation. The syntax and typing rules for System F_ω are given in figure 2.3.

Barendregt credits Girard with the development of System F_ω during his work in 1972 on his doctoral thesis. System F_ω is the kernel used by many modern compilers and is powerful enough to represent Haskell98 type classes [WB89] and GADTs [Atk12]; [VW10]. To see how type-classes are captured, consider that type-class constraints can be reduced to dictionary passing, or simply a collection of type-specific methods. For instance the function

```
Show a => a -> String
```

might reduce to

```
forall a. Show a -> a -> String
```

where **Show** is the collection of methods exposed by the **Show** typeclass. Type classes are captured The primary change from System F introduced

Kinds $\kappa ::= \star \mid \kappa \to \kappa$

Types $\tau ::= \lambda\alpha : \kappa.\tau \mid \forall\alpha : \kappa.\tau \mid \tau\ \tau \mid \tau \to \tau$

$\boxed{\Delta \vdash \tau : \kappa}$ (well-formed types)

$$\text{WfTyFn}\frac{(\alpha, \kappa_1), \Delta \vdash \tau : \kappa_2}{\Delta \vdash \lambda\alpha : \kappa_1.\tau : \kappa_1 \to \kappa_2}$$

$$\text{WfAll}\frac{(\alpha, \kappa), \Delta \vdash \tau : \star}{\Delta \vdash \forall\alpha : \kappa.\tau : \star}$$

$$\text{WfTyApp}\frac{\Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1\ \tau_2 : \kappa_2} \qquad \text{WfArrow}\frac{\Delta \vdash \tau_1 : \star \quad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \to \tau_2 : \star}$$

$\boxed{\Delta \vdash \tau \equiv \tau : \kappa}$ (type conversion)

$$\text{Refl}\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \tau \equiv \tau : \kappa} \qquad \text{Sym}\frac{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa}{\Delta \vdash \tau_2 \equiv \tau_1 : \kappa}$$

$$\text{Trans}\frac{\Delta \vdash \tau_1 \equiv \tau_2 : \kappa \quad \Delta \vdash \tau_2 \equiv \tau_3 : \kappa}{\Delta \vdash \tau_1 \equiv \tau_3 : \kappa}$$

$$\text{App}\frac{\begin{array}{c}\Delta \vdash \tau_1 \equiv \tau_1' : \kappa_1 \to \kappa_2 \\ \Delta \vdash \tau_2 \equiv \tau_2' : \kappa_1\end{array}}{\Delta \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa_2}$$

$$\text{Beta}\frac{(\alpha, \kappa_1), \Delta \vdash \tau_1 : \kappa_2}{\Delta \vdash (\lambda\alpha : \kappa_1.\tau_1)\ \tau_2 \equiv \tau_1[\alpha \mapsto \tau_2] : \kappa_2}$$

$$\text{Eta}\frac{(\alpha, \kappa_1), \Delta \vdash \tau : \kappa_2}{\Delta \vdash (\lambda\alpha : \kappa_1.\tau\ \alpha) \equiv \tau : \kappa_2}$$

Expressions $e ::= x \mid \lambda x : \tau.e \mid \Lambda\alpha : \kappa.e \mid e\ e \mid e[\tau]$

$\boxed{\Delta; \Gamma \vdash e : \tau}$ (well-formed expressions)

$$\text{WfConv}\frac{\Delta \vdash \tau \equiv \tau' : \star \quad \Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash e : \tau'} \qquad \text{WfVar}\frac{(x, \tau) \in \Gamma}{\Delta; \Gamma \vdash x : \tau}$$

$$\text{WfLam}\frac{(x, \tau_1), \Gamma \vdash e : \tau_2 \quad \Delta \vdash \tau_1 : \star}{\Delta; \Gamma \vdash \lambda x : \tau.e : \tau_1 \to \tau_2} \qquad \text{WfApp}\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{WfLamTy}\frac{(\alpha, \kappa), \Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha : \kappa.e : \forall\alpha : \kappa.\tau} \qquad \text{WfAppTy}\frac{\Delta; \Gamma \vdash \tau_1 : \kappa \quad \Delta; \Gamma \vdash e : \forall\alpha : \kappa.\tau_2}{\Delta; \Gamma \vdash e[\tau_1] : \tau_2}$$

Figure 2.3: System F_ω syntax and typing rules

by System F_ω is the addition of the *WfConv* rule which allows for terms to be coerced between provably equal types τ_1 and τ_2. While these declarative rules are a convenient means of specifying the system, they are not well suited to implementation on a machine. The crux of the problem lies in the potential use of the *WfConv* rule at any point and the term e to convert is in a sense too "bare," i.e. there is not enough information to give a syntax directed means of typing terms since a *WfConv* can potentially be inserted at any point. The usual solution to this problem is to transform the rules so that they can be applied in an algorithmic, syntax-directed way (see e.g. the introduction given by Weirich [Wei02] to making rules algorithmic, along with how the correctness of the transformation is proven) by removing *WfConv* and bundling a conversion check into the application rule since application sites are the only places requesting a particular type. Unfortunately, this is still problematic because checking type equality is unclear from the conversion rules. The solution to this final problem is usually to compute a term's "normal form," a form which is somehow as reduced as possible e.g. by repeated application of *Beta*. This transformation moving type conversion directly into the application rule can be seen in figure 2.4 of the dependently typed λ-calculus.

The work on the type and index expression systems of $\mathscr{L}$ developed in this thesis are most closely related to System F_ω exploring a system in which no explicit equations on syntactic forms are required. Additionally we take the approach of explicit applications of a *WfConv* style rule and show how the use of dependent types make such an approach tractable.

Despite the great expressive power of System F_ω, types can still only depend on types. By moving along to the upper right corner of the lambda cube $\lambda_{\mathbf{C}}$, the system is extended with types that can depend on terms, yielding full dependent types.

Dependent types

Dependent types extends the ideas of System F_ω to their conclusion, allowing for computation in types which depend on values. Curry did some of the earliest formalizations of dependent types as a combinatory logic [CF67], Coquand [CH88] who developed the calculus of constructions, and Per Martin-Löf [MS84] who developed Intuitionistic type theory. The usual rules associated with a dependently typed λ-calculus are given in figure 2.4. This type system while only adding value dependency in types, represents a great

Expressions $\quad e, t ::= \star \mid x \mid \lambda x : t.e \mid \Pi x : t.e \mid e\ e$

$\boxed{\Gamma \vdash e : t}$ (well-formed expression)

$$\frac{}{\Gamma \vdash \star : \star} \qquad \frac{(x,t) \in \Gamma}{\Gamma \vdash x : t}$$

$$\frac{(x,t_1), \Gamma \vdash e : t_2 \quad \Gamma \vdash \Pi x : t_1.t_2 : \star}{\Gamma \vdash \lambda x : t_1.e : \Pi x : t_1.t_2}$$

$$\frac{(x,t_1), \Gamma \vdash t_2 : \star \quad \Gamma \vdash t_1 : \star}{\Gamma \vdash \Pi x : t_1.t_2 : \star} \qquad \frac{\Gamma \vdash e_1 : \Pi x : t_1.t_2 \quad \Gamma \vdash e_2 : t_1' \quad \Gamma \vdash t_1 \simeq t_1'}{\Gamma \vdash e_1\ e_2 : t_2[x \mapsto e_2]}$$

Figure 2.4: Syntax and typing rules of the dependently typed λ-calculus.

.

simplification and departure from System F_ω. For instance we now have just a single syntactic category—that of terms—rather than distinct categories for values, types, and kinds. Instead we simply use the extra meta-variable t to express the intended interpretation of a term as either a "computational" expression or a type.

Another note to make is that the system presented includes the inconsistent rule $\vdash \star : \star$ subjecting it to Girard's paradox [Gir72]; [Coq86]. A possible solution necessitates complicating the system with a hierarchy of types $\vdash \star_i : \star_{i+1}$ [HP91]. These types are called "universes" and capture the idea that we might wish to define functions which operate over some entire *type*. For instance we might like to write a function which takes a list of types and produces an N-ary product, i.e. transporting $[A, B, C]$ to $A \times B \times C$. The list of such a function consists of types, i.e. $\star$; however it too must have some type. The type of such a type must be somehow "larger" than the types it contains, thus if A, B, and C are in $\star$ then $[A, B, C]$ must have type $\star_1$ to avoid Girard's paradox. The idea behind such a paradox is similar to Russel's paradox in set theory where the set of all sets not members of themselves produces the contradition $A \in A \iff A \notin A$.

To better understand how to create a simple, dependently typed lambda calculus—without a hierarchy of types—refer to Löh, McBride, and Swierstra [LMS10] who provide a thorough presentation using Haskell.

Due to the potentially complicated form types may take as was the

case in System F_ω, a notion of convertibility is introduced. However rather than include an explicit *WfConv* rule, here conversion is only allowed at application sites. This allows for a relatively simple type-checking scheme if term convertibility can be automated. Adapting Luo's definition of convertibility for the Extended Calculus of Constructions:

Definition 2.1.1 (reduction and conversion)**.** Two terms are convertible $e_1 \simeq e_2$ iff there exists some e_2' α-equivalent to e_2 obtained by performing a sequence of β expansions and contractions on e_1 where the β relation is

$$(\lambda x : t.e_1)\ e_2 \rightsquigarrow_\beta e_1[x \mapsto e_2]$$

In practice this definition allows for a relatively natural means of conversion checking by performing reduction on both sides and checking α equivalence.

Perhaps unsurprisingly, the size and complexity of heavily dependently typed terms grows rapidly. Modern languages with a focus on dependent types attempt to manage this complexity by providing high levels of automation and inductive data types which were first precisely formalized in Martin-Löf's type theory by Dybjer [Dyb94]. In Coq, the calculus of constructions was extended with inductive data types, yielding the calculus of inductive constructions (CIC) by Coquand [CP90]. However Coq does not provide much support for dependent pattern matching, thus often carefully hand-crafted elimination functions for walking over data-types and introducing values with the correct types are required. In our work the language Idris is used which is based on Luo's Universal Type Theory, often abbreviated as UTT. In the next section we present a discussion of the language Idris and how to use it.

2.2 Idris

Idris is a language with dependent types allowing for rich type-level computation and for terms to be indexed by arbitrary terms. Idris aims to support the development of real-world applications using dependent types. Idris, therefore, exposes support for (optionally unquantified, or "very") implicit arguments, pattern unification, partiality, ad hoc function overloading, typeclasses, and elaborator reflection [CB16] which will be briefly covered in this section along with an introduction to the language itself. For a full exposition

on the design, implementation, and usage of Idris see Brady [Bra13b] or refer to the recent book on type-driven program development with dependent types by Brady [Bra16].

As presented by Brady [Bra14a], Idris at its core mostly follows the specification of dependent types given in figure 2.4. To avoid Girard's paradox as described in section 2.1 on page 26 Idris uses a hierarchy of types. However to avoid the tedium of specifying the exact level of each type defined, Idris using constraint solving [HP91] to figure out which level each type should exist at. Thus in Idris program expressions have types, and types have types, without any need to explicitly mention levels, kinds, etc.

2.2.1 Programming with Idris

Following the Curry-Howard correspondence [How80]; [CF67] in Idris one way of expressing propositions is to define a series of relations over terms by constructing a new type. For example given the usual definition of peano numbers having type `Nat`, even parity can be captured by the following type

```
data IsEven : Nat -> Type where
  ZeroEven : IsEven Z
  SucSucEven : IsEven n -> IsEven (S (S n))
```

The line `data IsEven : Nat -> Type where` declares a new type `IsEven` which is indexed by a natural number and inhabits the universe `Type`. This type captures the following judgments that zero is even and that if a number n is also, then so is the number $n + 2$[1].

$$\text{ZeroEven}\frac{}{\mathit{IsEven}\ 0} \qquad \text{SucSucEven}\frac{\mathit{IsEven}\ n}{\mathit{IsEven}\ (n+2)}$$

Constructing a proof of a given number's even-ness is reduced to finding a sequence of `IsEven` constructors. For example to show that four is even it can be shown that zero is even, and that zero plus two is even, and finally that because zero plus two is even, zero plus two plus two is even.

```
total fourEven : IsEven 4
fourEven = SucSucEven (SucSucEven ZeroEven)
```

[1]Note that we assume that addition is defined recursively, i.e. $(n+1)+m = (n+m)+1$ and as a notational convenience may write constants on the right as in $n + 2$ when in reality a theorem prover might only reduce the term $2 + n$.

Note the decorator `total` attached to the definition `fourEven`. This is necessary because Idris supports *partial* functions, i.e., functions which are not defined for every input because e.g., they diverge or merely have a definition omitted. These annotations are necessary because Idris must give special handling to partial functions or face possible divergence during type-checking. To avoid this, Idris will only automatically expand partial functions by their definition one level at a time. Note that within this work all functions presented are total and thus the `total` keyword is generally elided.

What about programmatically checking if a number is even? Answering this question involves crafting a *decision procedure* which is simply an algorithm which in this case describes if its input is even or odd. In the case of an even input, a certificate of type `IsEven` will be produced, what if the value is odd? In this case the decision procedure can answer with a proof that were we to *assume* that the value were proved even—by producing a value of type `IsEven`—a contradiction can be derived. In Idris this type of decidability of a proposition is called `Dec` and exposes two constructors `Yes` and `No`. If a proposition holds then `Yes` takes a proof of the proposition. If a proposition does not hold then `No` provides a proof that a proposition holds and requests that a contradiction be derived. Using `Dec`, deciding the parity of a number can be done inductively, breaking the problem down into three cases

1. If the input is zero, it's evidently even.
2. If the input is one, it's evidently odd.
3. If the input is of the shape $n + 2$, then if n is even, by the `SucSucEven` rule $n + 2$ is even; however if n is odd $n + 2$ must odd otherwise we have that n is odd and even.

This reasoning is relatively directly transcribed into Idris

Listing 2.1: Decision procedure for number parity

```
isEven : (n : Nat) -> Dec (IsEven n)
isEven Z = ZeroEven
isEven (S Z) = No (\oneIsEven => ?contr)
isEven (S (S n)) with (isEven n)
  isEven (S (S n)) | Yes nEven  = Yes (SucSucEven nEven)
  isEven (S (S n)) | No notEven = No (\(SucSucEven nEven) =>
                                        notEven nEven)
```

The above definition presents several features of Idris

1. *Program holes*: the term `?contr` is a program hole which aids in program development by refinement.

2. *with Expressions* which allow us to perform dependent pattern matching, in turn learning something about the input and helping Idris perform reduction of the goal. This will be further elaborated on in section 2.3.1.

Program holes are a crucial feature for development of dependently typed programs. The hole `?contr` represents an unwritten term, allowing the type-checker to assert that the *context* surrounding the hole is valid. This allows for programs to be written by refining pieces of an expression and checking that—provided some assumptions in the form of holes hold—those individual pieces hold. Here this means that the term `No`, and its single argument of a function to `?contr` is correct, and thus the final proof need only fill in the body of the hole. Here the type of `?contr` is

$$\frac{\mathit{oneIsEven} : \mathit{IsEven}\ 1}{?\mathit{contr} : \mathit{Void}}$$

However the reason the value `oneIsEven` of type `IsEven : 1` can't possibly hold is due to there being no inhabitant of `IsEven` indexed by `1`. Thus the value `oneIsEven` can't be matched upon. In order to aid in proving these cases Idris provides a special keyword `impossible`.

Proving contradictions

Idris supports the proving of branches in which pattern matching is impossible by the *absurd pattern*. When pattern matching is impossible, a branch in Idris can be marked impossible and no body need be given. This is particularly useful in proving negative clauses where a value with an obviously empty type is assumed. Consider the case of proving that one is not even, given an assumption that it is even

```
oneIsNotEven : IsEven (S Z) -> Void
oneIsNotEven pf impossible
```

Instead of providing a program body, the clause is marked `impossible`. In this case Idris will check that the possible expansions of the term `pf` are invalid. To understand, performing expansion by hand `pf` is either `ZeroEven`

or `SucSucEven`. In the former case `pf` would have type `IsEven Z` and in the latter case `pf` would have type `IsEven (S (S n))`; however the concrete type of `pf` is `IsEven (S Z)` and thus by discriminating the value of the index it's clear that no possible pattern of `pf` will produce a branch that can be reached. Note that by pattern matching on the `pf` term something is *learned* about the shape of the terms it relates to—in this instance something is learned about the number n it's indexed by. The notion of forcing a value by refinement through pattern matching is at the heart of dependent types. Indexed terms can have their shape restricted by ensuring the index has a particular value, and similarly by matching on indexed terms the values of their indices can be restricted. By performing pattern matching these value restrictions are pushed into the type system.

The above construction of even-ness is called a *view* [MM04b]. Value refinement by pattern matching on views is an important strategy for theorem proving with dependent types.

2.3 Views, existentials, and equality.

Views are useful because they provide a different means of reasoning about some underlying value. Instead of proving properties about a number by reasoning about it directly, the view itself can be reasoned about instead. In the case of an even number n this means that it's immediately evident that n can be written as $n = m + 2$ rather than by appealing to some auxiliary well-formedness constraint. Consider the proof of divisibility of an even number n by decomposition into the product $2 * m$. A simple inductive proof might be sketched by noting that in the zero case $n = 0 = 2 * 0$, and in the inductive case there's some m such that $2 * m = n$ reducing to the final obligation of finding an m' such that $2 * m + 2 = 2 * m'$. Clearly taking $m' = m + 1$ gives $2 * m + 2 = 2 * (m + 1) = 2 * m + 2$ as required.

To prove this in Idris requires notions of existentially quantified variables and equality. Since exists is a type of variable binder, it's unsurprising that properties over existentials are encoded as an arrow in Idris

```
data Exists : (a : Type) -> (prop : a -> Type) -> Type where
  Witness : (x : a) -> p x -> Exists x p
```

The above type simply asserts that to prove there exists a value with some property `prop`, it's enough to present a concrete witness x and a proof that the desired property p holds on that value x. Idris has a special notation for

existentials which is covered thoroughly in the next section: the proposition `Exists Nat (\x => p x)` is written `(x : Nat ** p x)`, and its inhabitant `Witness x px` is written `(x ** px)`.

2.3.1 Notions of equality

Term equality in Idris is written in the usual way `x = y` and the proof that a term equals itself is written `Refl` which is the judgment that every term equals itself, i.e. `Refl : x = x`. In Idris for example this admits proofs of all the properties that can be expected of an equivalence relation

```
data (=) : a -> a -> Type where
  Refl : x = x
symmetry : x = y -> y = x
symmetry Refl = Refl
transitivity : x = y -> y = z -> x = z
transitivity Refl xEqz = xEqz
```

The above definition of equality in Idris `(=)` is called *propositional equality* because it brings the definitional equality of the underlying theory—i.e. that every term is equal to itself—to the developer. In Idris two terms are considered to be definitionally equal when they reduce to the same normal form.

Note the simplicity of the symmetry and transitivity proofs by pattern matching on the identity proofs. In the case of symmetry by matching on `Refl` the system learns that y *is* x and thus the proof obligation becomes a simple case of showing that $x = x$. Similarly to prove `transitivity` we first match on the proof that $x = y$ allowing Idris to identify the two converting the type of the second term to $x = z$ which satisfies the newly rewritten obligation. Alternatively a second match of `Refl` could be performed requiring that the trivial equality $x = x$ be proven. The assumption that identity proofs are unique requires an axiom dubbed K stating $\forall p : x = y, p =$ Refl i.e. that any proof p that $x = y$ is equal to the proof Refl. Although Idris includes this axiom it has been shown that pattern matching is recoverable without K [CDP14]. Note that in Idris the equality operator `(=)` is overloaded and may instead refer to *heterogeneous equality*. Heterogeneous equality—sometimes called John Major equality [McB00]—captures the idea that a value equals itself even at different types

```
data (=) : a -> b -> Type where
```

```
Refl : x = x
```

To see the utility of such a definition consider some length n list `xs` of type `Vect n Nat` and the following proposition `xs = xs ++ []` which states that appending nil on the right of `xs` equals `xs`. Despite the obvious truth of this assertion, without heterogeneous equality the mere statement will fail to pass type-checking because the types on the left and right-hand terms are unequal. Consider that `xs : Vect n Nat` but that `xs ++ [] : Vect (n + 0) Nat`. Heterogeneous equality solves this problem. For a more complete coverage of heterogeneous equality and its usage refer to McBride [McB00] and Hur [Hur10].

Equipped with existentials and equality the simple theorem that an even number can be decomposed into a product of two can be proven.

```
decomp : (n : Nat) -> IsEven n -> (m : Nat ** n = 2*m)
decomp Z         ZeroEven = (Z ** Refl)
decomp (S (S n')) (SucSucEven nEven) with (decomp n' nEven)
  decomp (S (S (S (S Z)*m))) (SucSucEven nEven)
  | (m ** Refl) = (S m ** Refl `trans` sucCommutes)
```

Following the outline of the proof sketch the first case $n = 0$ is trivial with $m = 0$ since Idris reduces $2 * 0$ to 0. The inductive case of $n = 2 + n'$ first decomposes n' into $2 * m$ by applying the **`with`** statement and matching on the result. Note that by matching on the resulting equality proof, we learn that $n' = 2 * m$, changing the pattern on the left-hand side from `S (S n'))` to `S (S (S (S Z)*m))`. This matching reduces the proof obligation to $\exists m'.2 * m + 2 = 2 * m'$ which from the sketch clearly holds for $m' = m + 1$. The proof of this resulting equality in Idris appeals to one final lemma about the commutativity of successor, i.e. $2+(m+(m+0)) = (1+m)+((1+m)+0)$.

```
sucCommutes : S (S (n + (n + 0))) = S n + (S n + 0)
```

One curiosity about the above propositions and proofs is the occurrence of seemingly unbound variables such as n in `sucCommutes`. These are *implicits* which expand the notion of a hole to wider contexts. In `sucCommutes` the quantification of the parameter $n : \mathbb{N}$ is clearly not only inferred at the definition, but additionally the *argument* in n passed in the application of `sucCommutes` in `decomp` is inferred. In this case Idris must solve for the appropriate value of n which in this case is $n = m$. In fact Idris can be asked to solve for arbitrary values by placement of an underscore. The next section gives a presentation of the programming features Idris exposes around holes, implicits, and underscores.

Holes and implicit arguments

As is evident in dependently typed languages due to the heavy indexing of types there are a great number of auxiliary parameters to types and functions that should be inferred by the type-checker and therefore can be omitted at definition and call sites. To explicitly quantify implicit parameters, they are wrapped in curly braces. For example the fully explicit definition of length indexed vectors can be written as

```
data Vect : Nat -> Type -> Type where
  Nil : {a : Type} -> Vect Z a
  (::) : {a : Type} -> {n : Nat} -> a -> Vect n a
      -> Vect (S n) a
```

To explicitly pass a value in an implicit position or to match on an implicit parameter such as n similar syntax is, wrapping the parameter name in curly braces as in `{n=Z}`. For example the fully explicit definition of vector append can be written

```
append : {a : Type} -> {n, m : Nat} -> Vect n a -> Vect m a
      -> Vect (n + m) a
append {a=a} {n=Z}    {m=m} []         ys = ys
append {a=a} {n=S n'} {m=m} (x :: xs) ys =
  append {a=a} {n=n'} {m=m} xs ys
```

This illuminates the need for an automated means of treating implicit parameters and arguments to remove unnecessary noise from definitions and their proofs. In addition to inferring implicit arguments, Idris can infer arbitrary values by placement of an underscore.

In Idris implicits are closely related to the notion of *erasability*, or terms which carry no necessary computational content and can therefore be removed as dead-code during compilation. Intuitively this is the case because if a term can be inferred by some automated procedure, then all the information it carries should be recoverable from the context. This style of optimization was explored early on by Brady, McBride, and McKinna [BMM03]. Erasure plays an important role in making dependently typed code efficient and has previously been explored by Luther [Lut03]. Recently Idris has gained support for marking values as to-be-erased using a dot, for example the definition of even could mark its index as erasable

```
data IsEven : Nat -> Type where
  ZeroEven : IsEven Z
  SucSucEven : .{n : Nat} -> IsEven n -> Even (S (S n))
```

This would ensure that at runtime only the proof would be used for computation. Explicit dots are usually not needed as Idris performs a whole program usage analysis to decide which terms are erasable, and which are not. An extensive discussion of Idris' aggressive, flexible erasure features is given in Tejiščák [TB15]

Some dependently typed languages such as Coq [Her05] expose two universes of types: `Set` the type of unerased proofs, and `Prop` the type of erasable proofs. The system then ensures that no terms from `Prop` are used in computations of terms in `Set`. Others such as ATS [Xi17] the successor of DML [Xi07] require that all proofs be erasable. This distinction of where proofs live gives rise to a stratified classification of dependently typed systems.

2.4 Languages with dependent types

The existing space of languages with dependent types is large, with work dating back to Haskell B Curry's writing on a combinatory logic [CF67] which includes the Ξ operator acting like a π binder. A number of the key implementations have been listed in table 2.1[2] which will be briefly covered. Although dependently typed systems differ in their underlying theory and capacity for working with complicated interdependent indices, languages can largely be broken into three categories.

1. Full-spectrum dependent types: types can depende on values with no distinction between runtime and compile time arguments.

2. Dependent types: types can depend on values, however there is a clear phase distinction and types cannot take part in runtime computations.

3. Synthetic dependent types: a synthesized form of dependent types without a true pi binder. Synthetic dependent types use existing types and type-level functions to mimic dependent types.

Each of the above styles of dependent types offer a number of unique trade-offs.

[2]Coq is not strictly speaking proven correct, although work has been done by Barras and Werner [BW97] to extract a Caml program for checking CoC terms.

Language	EDSL	Dependent Types	Verification Language
$\mathscr{L}$	Yes	Synthetic	Idris
Ωmega [SL07]	No	Synthetic	
LXRes [CW00]	No	Synthetic	
Haskell [Com10]	No	Synthetic	
ATS2 [Xi17]	No	Yes	ATS2
Twelf [PS99]	No	Yes	
Coq [Tea18]	No	Full-Spectrum	Coq
Matita [Asp+06]	No	Full-Spectrum	
Epigram [McB04]; [MM04b]	No	Full-Spectrum	
Cayenne [Aug98]	No	Full-Spectrum	
Lean [Mou+15]	No	Full-Spectrum	
F* [Str+12]	No	Full-Spectrum	Coq/F*
Blodwen [Bra18]	No	Full-Spectrum	Idris
Idris [Bra13b]	No	Full-Spectrum	
Agda [Nor08]	No	Full-Spectrum	

Table 2.1: Languages with dependent types

Full-spectrum dependent types

Full-spectrum dependent types are the "familiar" notion of dependent types presented in figure 2.4. Here types can freely move between concrete expressions and types. While this flexibility is convenient for writing and verifying programs, it creates a unique set of problems which must be solved in order to generate efficient programs. In particular it can be useful to match on an indexed term in order to perform some evaluation. However such a term can be much larger than the concrete data indexing it, sometimes *exponentially* so. Worse still, this proof will be kept around at runtime by a naive compiler making for example an otherwise linear time operation exponential. Thus a performance conscious compiler for a dependently typed language should erase indexed terms uniquely determined by the context in which they appear. To exemplify this problem Tejiščák and Brady [TB15] give the following definition of what it means for a list to be a palindrome.

```
data Palindrome : List Nat -> Type where
```

```
PNil : Palindrome []
POne : (x : Nat) -> Palindrome [x]
PTwo : (x : Nat) -> Palindrome xs
    -> Palindrome (x :: xs ++ [x])
```

In the above `PNil` asserts that the empty list is a palindrome, `POne` asserts that the singleton list is a palindrome, and `PTwo` asserts that if `xs` is a palindrome then placing a value at both ends of the list is still a palindrome. In order to check if an arbitrary list of nats is a palindrome, a straightforward strategy is to introduce a view of a list exposing its head and tail simultaneously rather than having a cons exposing just the head. This is the type of "V-lists" given in the Idris prelude

```
data VList : List Nat -> Type where
  VNil : VList []
  VOne : VList [x]
  VTwo : {x, y : Nat} -> .{xs : List Nat} -> VList xs
      -> VList (x :: xs ++ [y])
```

Checking that a given V-list is a palindrome is of course trivial. Simply walk along spine testing if $x = y$ at each step. In the above definition clearly the arguments `xs` is fully determined by the recursive argument of type `VList xs`. Thus the list indexing the `VList` should not be kept at runtime because it's not needed to perform computation. This is indicated to the Idris compiler by dotting the argument `xs`.

Most of the languages in table 2.1 expose a theory with full-spectrum dependent types although they differ in underlying feature set. Surprisingly the only modern dependently typed language with a formally verified core is F* whose type-checker is bootstrapped in Coq. Although the type-checkers of Idris and Coq are not formally verified, they are implemented as a small kernel which should—at least philosophically—increase trust.

Dependent types

Dependently typed systems are sometimes not qualified with the term *full-spectrum*. Though not always, this often indicates that the system has a strict, well-defined, compile-time phase distinction serving to separate the universes of static, compile-time terms and dynamic run-time terms. A primary advantage of this approach is that type erasure becomes trivial, guaranteeing the structure of a proof about a term does not affect the performance of computations which use the term. This restriction is double-

edged because it is sometimes the case that a proof term greatly simplifies the implementation of an algorithm. For example the type of list element membership proofs carries solely the information required to perform e.g. a lookup, and thus the distinction of proof terms and runtime terms is moot. The theorem prover Twelf [PS99] and the high-performance system's language ATS2 [Xi17] are examples of languages implementing this restricted style of dependent types. These languages roughly correspond to $\lambda\mathbf{P}$ in Barendregt's cube. The language ATS2 in particular has taken advantage of the performance afforded by a clear phase distinction having developed formally verified Linux kernel modules and programs with throughput and latency characteristics in line with C. Finally ATS2 is written in ATS2 with large pieces of correctness formalized. This is trivially possible because ATS2 emits C, and thus a bootstrap compiler can be distributed.

Synthetic dependent types

The term *synthetic dependent types* was coined by Crary and Weirich [CW00] during their development of a language based on LXRes [CW99] for analyzing and verifying program costs using types. The intuition behind the system was that a program's runtime cost could be measured by performing a symbolic evaluation in the context of an extra "clock" parameter. The clock type $m : \mathbb{N}$ would count down towards a timeout value and thus the runtime cost of a program can be measured by instantiating the clock with the precise number of "ticks" required. Hence each function takes and returns a pair where the first projection is the usual type, and the second projection is the current clock value. For example an addition function taking 1 tick satisfies the type $(\mathbb{N} \times \mathbb{N}, 1) \to (\mathbb{N}, 0)$ accepting the two naturals to sum and using up one clock tick. To apply this in a more general context the function should accept the current clock value as an argument and reduce it by 1 i.e. $\forall n : \mathbb{N}.(\mathbb{N} \times \mathbb{N}, n+1) \to (\mathbb{N}, n)$. Of course such an analysis requires symbolic information about its program terms which would usually be provided by a system with dependent types. However in order to target simpler systems such as Typed Assembly Language [Cra03]; [Mor+98] which lack dependent types, a close approximation can be had by considering System F style types with a sufficiently complicated space of kinds. For instance, consider a system with two kinds: *Nat* and *Type*. Weirich and Crary give an example of a how to describe the cost of a higher-order `map_pair` function which applies some f to both elements of a pair. If the cost of f is k, and the

overhead of a function call is 1, then the cost of applying f twice must be $2k + 2$. This can be typed using a clock parameter by reducing $n + 2k + 2$ to n.

$\Lambda\alpha : \mathit{Type}, \beta : \mathit{Type},$
$\quad \lambda(fp : (\forall m : \mathbb{N}.(\alpha, m + k) \to (\beta, m)) \times (\alpha \times \alpha), n + 2k + 2).$
$\quad \text{let } f = \text{prj}_1\ fp \text{ in}$
$\quad \text{let } p = \text{prj}_2\ fp$
$\quad \text{in}(f[n + k + 1](\text{prj}_1\ p), f[n](\text{prj}_2\ p))$

From the type signature we derive the starting clock value of $n+2k+2$ with the goal of reducing the clock to n, thus using up $2k + 2$ ticks as expected. The first call to f has type $f[n+k+1] : (\alpha, (n+k+1)+k) \to (\beta, n+k+1)$, taking up k ticks; however the overhead of the call itself takes an additional tick so the remaining clock value of $n + k + 1$ is reduced to $n + k$. Finally the second call has type $f[n] : (\alpha, n + k) \to (\alpha, n)$ thus expecting a clock value of $n + k$ which was just shown to hold.

This type system is extended with support for sums, products, and fixpoints of kinds admitting analysis of a wide variety of data structures. To see how a sort of synthetic dependent type is recovered consider a binary tree given as a sum type where the left-hand side is the leaf constructor and the right-hand side is the branch constructor

```
type Tree = unit + (Nat * Tree * Tree)
```

A suitable type-level encoding of such trees is a declaration of a new kind

```
kind TreeRep = Leaf | Node of TreeRep * TreeRep
```

Then the `Tree` type should be rewritten to accept a `TreeRep` as an index where the `Leaf` kind is only inhabited by leaves (left-hand of the sum) and the `Node` kind is only inhabited by branches (right-hand of the sum)

```
type Tree(s) =
  (case s of
    | Leaf => unit
    | Node _ => void) +
  (case s of
    | Leaf => void
    | Node (s1, s2) => Nat * Tree(s1) * Tree(s2))
```

Thus if `s=Leaf` then attempting to construct a node (right-hand side) will require the programmer to produce a value of type `void` which is of course not possible. The case of `s=Node` is symmetric. Thus the static value of `s` captures the runtime value of its underlying tree by a suitable restriction on the possible constructors that can legally be applied.

Despite the name synthetic dependent types are curiously not an implementation of any dependent type theory, but more a clever deployment of a theory reminiscent of System F_ω. Still synthetic dependent types form the foundation of the work in this thesis, being both simple and expressive while admitting strong properties such as checkable erasability. In contrast the implementation of the system described in this thesis is within Idris, touting full-dependent types. While the possibility of this embedding is not terribly surprising given the obviously reduced expressivity of synthetic dependent types, the next section covers a basic introduction to the use of dependent types to capture simpler type systems such as the simply typed λ-calculus. We explore both the traditional construction of correctness as well as how dependent types can greatly reduce the burden of proof obligations required by type-safety.

2.5 Mechanized verification of type-safety

As a first illustration of how internalizing correctness properties using dependent types simplifies specific proofs a formalization of the simply typed λ-calculus (STLC) is given. Recall the types and syntax of the STLC from figure 2.1

$$\tau ::= Nat \mid \tau \to \tau$$
$$e ::= n \in \mathbb{N} \mid x \mid e + e \mid \lambda x.e \mid e\ e$$

The safety property of interest relates to the soundness of term *evaluation*. More specifically, the evaluation strategy of interest is called *weak-head-normal-form reduction* (whnf). Here a term is considered to be evaluated to weak head normal form if it is a constant value, or an unapplied function. In the style of Wright and Felleisen [WF94] subject reduction is considered to be correct if every well-formed term evaluates to a value, the resulting value is well-formed, and has retained its type. To make this notion rigorous requires a rewriting *operational semantics* which we will now discuss.

2.5.1 Structural operational semantics

Operational semantics of a language refers to a rewrite system which is at its heart a bag of rules specifying how to rewrite various input strings to well-defined output strings. Somewhat more rigorously, operational semantics consist of some *evaluation relation* given directly on the abstract syntax of the language. Operational semantics usually refers to either *small-step* or *big-step* semantics. In small step, or structural operational semantics a *step relation* $e \longrightarrow e'$ is defined, asserting that the input string e can be rewritten to the output string e'. Much of the early work on structural operational semantics is attributed to Plotkin [Plo75]; [Plo04b]; [Plo04a] and Felleisen [FH92]; [WF94]. Operational semantics is a convenient means of describing a program's execution because it closely captures the sequential style of bare-metal machines whereby e.g. a loop is expressed as the repeated rewrite of its body.

A thorough introduction to operational semantics and its applications to proving soundness properties of type systems can be found in Pierce [Pie02].

Given that small step operational semantics performs just a single step of evaluation at a time, this often results in a large number of rules e.g. to evaluate addition expressions there must be a rule to step the addend, to step the augend, and finally to reduce the addition of canonical values to their arithmetic sum. Although strategies exist to reduce this burden such as the usage of contexts which pull out a single reducible expression (redex) from a term, an alternative approach is to use big-step operational semantics.

2.5.2 Natural operational semantics

Early work on big step operational semantics, also called natural semantics is attributed to Kahn [Kah87]. Kahn noted that small step semantics requires a great deal of machinery that denotational semantics does not and proposed a relational specification of evaluation where program terms are reduced to values in a single step. The step relation $e \Downarrow v$ asserts that e evaluates to the value v. Big step operational semantics is convenient because it closely matches the way that an evaluation might be written in a functional language, mapping each legal syntactic term to a resulting value. One complication of big step semantics requires the use of a counter to differentiate between *stuck* terms and *divergent* terms. To ensure that a program terminates reduces to

finding some finite n such that $e \Downarrow_{n+1} v$. Nielson and Nielson [NN07] offer a thorough introduction to big step semantics.

The evaluation of the STLC is readily modeled using big-step semantics. In the next section this semantics is discussed, along with details of how to encode these ideas using Idris, and how the use of dependent types can produce a simpler, alternative, verified encoding.

2.5.3 Mechanized type-safety of STLC

Wright and Felleisen [WF94] developed a notion of a language's type-safety using a *small-step operational semantics*, breaking the concept down into two properties: *progress* which states that if a term is well typed, and is not a value, then it can always perform a step of computation

$$\Gamma \vdash e : \tau \implies (\exists e'.e \longrightarrow e') \vee (\exists v.v = e)$$

and *preservation* which states that if a term is well-typed, then performing a step of computation results in a well-typed term

$$\Gamma \vdash e : \tau \wedge e \longrightarrow e' \implies \Gamma \vdash e' : \tau$$

In contrast, here the safety of the STLC is proven using a natural semantics in a single theorem, asserting that if a term is well-formed then evaluation will always produce some well-formed value of the same type. Thus strictly speaking progress and preservation don't make sense to apply to the big-step definition, however the ideas inform what equivalent notions we should be interested in proving.

A naive encoding of STLC syntax in Idris exactly transcribes the syntactic forms

Listing 2.2: STLC AST and types

```
data Ty : Type where
  TyNat : Ty
  TyArrow : Ty -> Ty -> Ty
data Stlc : Type where
  Lit : Nat -> Stlc
  Var : String -> Stlc
  Add : Stlc -> Stlc -> Stlc
  Lam : String -> Stlc
  App : Stlc -> Stlc -> Stlc
```

To evaluate STLC terms, they must satisfy the well-formedness condition from figure 2.1—otherwise e.g. an ill-formed term might try to add an unapplied function to a number. Recall well-formedness is defined with respect to a context and that well-formed variables are members of this context. The encoding of a context is a simple list of variable names and their types.

```
Ctx : Type
Ctx = List (String, Ty)
```

Membership is captured by a constraint on the result of attempting to lookup a variable in the context i.e. `lookupVar x ctx = Just t`.

```
lookupVar : String -> List (String, a) -> Maybe a
lookupVar x ((y, v) :: env) = if x == y then Just v
                                  else lookupVar x env
lookupVar x [] = Nothing

data WfStlc : Ctx -> Ty -> Stlc -> Type where
  WfLit : (n : Nat) -> WfStlc ctx TyNat (Lit n)
  WfVar : lookupVar x ctx = Just t -> WfStlc ctx t (Var x)
  WfAdd : WfStlc ctx TyNat e1 -> WfStlc ctx TyNat e2
       -> WfStlc ctx TyNat (e1 `Add` e2)
  WfLam : (x : String) -> (t1 : Ty)
       -> WfStlc ((x, t1) :: ctx) t2 e
       -> WfStlc ctx (t1 `TyArrow` t2) (Lam x e)
  WfApp : WfStlc ctx (t1 `TyArrow` t2) e1
       -> WfStlc ctx t1 e2
       -> WfStlc ctx t2 (e1 `App` e2)
```

In the above presentation of STLC, terms will evaluate to either a natural number, or a *closure*. Closures encode weak head normal functions by capturing the unnormalized function body along with the current mapping of free variables to values.

```
data Value : Type where
  VLit : Nat -> Value
  VClo : String -> Ty -> Stlc -> List (String, Value) -> Value
Env : Type
Env = List (String, Value)
```

The big step semantics of STLC terms is given in figure 2.5 Constants evaluate to constants. Variables evaluate to their mapping in the current evaluation environment. Lambda evaluate to closures capturing the current mapping of free variables. Addition expressions evaluate to the sum of their

Values $v ::= n \in \mathbb{N} \mid \lambda x : \tau.e$

$\boxed{\Gamma \vdash v : \tau}$ (well-formed values)

$$\text{WfVLit}\frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \text{WfClo}\frac{(x, \tau_1), \Gamma \vdash e : \tau_2 \quad \Gamma \vdash \rho}{\Gamma \vdash \lambda_\rho x : \tau_1.e : \tau_1 \to \tau_2}$$

$\boxed{\Gamma \vdash \rho}$ (well-formed environments)

$$\text{WfVLit}\frac{}{\Gamma \vdash \emptyset} \qquad \text{WfVLit}\frac{\Gamma \vdash v : \tau \quad \Gamma \vdash \rho}{(x, \tau), \Gamma \vdash (x, v), \rho}$$

$\boxed{e \Downarrow_\rho v}$ (big step semantics of expressions)

$$\text{EvalLit}\frac{}{n \Downarrow_\rho n} \qquad \text{EvalVar}\frac{(x, v) \in \rho}{x \Downarrow_\rho v}$$

$$\text{EvalLam}\frac{}{\lambda x : \tau.e \Downarrow_\rho \lambda_\rho x : \tau.e} \qquad \text{EvalAdd}\frac{e_1 \Downarrow_\rho v_1 \quad e_2 \Downarrow_\rho v_2}{e_1 + e_2 \Downarrow_\rho v_1 + v_2}$$

$$\text{EvalApp}\frac{e_1 \Downarrow_\rho \lambda_{\rho'} x : \tau.e \quad e_2 \Downarrow_\rho v_2 \quad e \Downarrow_{(x, v_2), \rho'} v}{e_1\ e_2 \Downarrow_\rho v_3}$$

Figure 2.5: Big step operational semantics of the STLC

evaluated arguments. Finally evaluation of applications is defined when the applied expression e_1 evaluates to a closure whose wrapped expressions is evaluated in the captured environment ρ' extended with the evaluation of the argument e_2. Defining evaluation as a relation, rather than a function, allows for the unchanged semantics to be given in Idris

Listing 2.3: STLC big-step evaluation relation

```
data Eval : Env -> Stlc -> Value -> Type where
  EvalLit : Eval env (Lit n) (VLit n)
  EvalVar : lookupVar x env = Just v
         -> Eval env (Var x) v
  EvalAdd : Eval env e1 (VLit n1)
         -> Eval env e2 (VLit n2)
         -> Eval env (e1 `Add` e2) (VLit (n1 + n2))
  EvalLam : Eval env (Lam x t e) (VClo x t e env)
  EvalApp : Eval env e1 (VClo x t e1' env')
         -> Eval env e2 v2
         -> Eval ((x, v2) :: env') e1' v1
```

```
          -> Eval env (e1 `App` e2) v1
```

Recall that type-safety ensures that every well-typed input term evaluates to a well-formed value, respecting the type of the input. The well-formedness of values given in figure 2.5 again can be directly encoded in Idris

```
mutual
  data WfValue : Ty -> Value -> Type where
    WfVLit : WfValue TyNat (VLit n)
    WfClo : WfStlc ((x, t1) :: ctx) t2 e
         -> WfEnv ctx env
         -> WfValue (TyArrow t1 t2) (VClo x t1 e env)
  data WfEnv : Ctx -> Env -> Type where
    WfNil : WfEnv [] []
    WfCons : WfValue t v
          -> WfEnv ctx env
          -> WfEnv ((x, t) :: ctx) ((x, v) :: env)
```

Again since this presentation uses a big-step semantics, progress and preservation aren't applicable. However in the big-step world the equivalent notions are a guarantee that every well-formed input is related to a well-formed output value, and that this value retains the type of the input. This allows for the following "fused" definition of "type-safety"

$$\Gamma \vdash \rho \wedge \Gamma \vdash e : \tau \implies \exists v.e \Downarrow_\rho v \wedge \Gamma \vdash v : \tau$$

In Idris this definition can be presented using the well-typing relations defined above.

```
evalSafe : WfEnv ctx env
        -> WfStlc ctx t e
        -> (v : Value ** (Eval env e v, WfValue t v))
```

The mechanized proof of this property requires a few particularly awkward definitions. Consider first the case of a variable. The obligation is to provide a value v along with proofs that v is well-typed and that it can be derived from e within the big-step relation i.e. a trace of the evaluation must be produced. By assumption e is well-typed and therefore we know that a lookup within the context will succeed by the `WfVar` rule. Unfortunately, transporting this result to the environment is tedious as we will show it requires manual zeta-expansion and contraction. Consider first the transcribed obligation

```
lookupSafe : WfEnv ctx env -> lookupVar x ctx = Just t
          -> (v : Value **
              (lookupVar x env = Just v, WfValue t v))
```

From the well-formedness of the environment and x it is immediately clear the environment must be non-empty for otherwise the lookup of x in the context would fail.

```
lookupSafe {ctx=[]} {env=[]} WfNil Refl impossible
```

However, expanding this environment leaves the following obligation in the non-empty case: `lookupVar x ((y, v') :: env) = Just v`. Clearly this holds because if x is y then we can show that v' is v, otherwise by induction x can be found somewhere in the tail `env`. Unfortunately on a machine these steps must be spelled out. First the definition of `lookupVar` must be reduced which will be blocked on the conditional statement over `x == y`. To reduce this requires the use of the "inspection pattern" which allows for an expression to be evaluated while remembering what the result of the expression was.

Listing 2.4: The inspection pattern

```
data Inspect : a -> Type where
  Remember : (y : a) -> x = y -> Inspect x
```

As an example of its use with the string equality test `x == y`

```
case inspect (x == y) of
  Remember True xEy  => ...
  Remember False xNy => ...
```

Where `xEy : x == y = True` and `xNy : x == y = False`. These proofs can be used to hand reduce the obligation `lookupVar x env = Just v` for example by appealing to an auxiliary function

```
reduceLookup : (x == y) = True
            -> lookupVar x ((y, v) :: env) = Just v
reduceLookup {x=x}{y=y}{v=v}{env=env} xEy =
  replace {P=\b => (if x == y then Just v
                       else lookupVar x env) =
                   (if b then Just v
                       else lookupVar x env)}
          xEy
          (Refl {x=if x == y then Just v
                     else lookupVar x env})
```

Worse still, in the case x and y are unequal and the result v is found in the tail of the environment a proof that `lookupVar` succeeds will be available, i.e. `lookupVar x env = Just v` while the requirement is to show

`lookupVar x ((y, w) :: env) = Just v`. Moving between these two proofs, while obvious, is tedious.

The problems encountered here are somewhat unique to a language where theorem proving is supposed to occur wholly within the usual programming paradigms. In particular, the difficulty in proving and reading terms in the classical style is due to the focus on the shapes of proofs. Traditional provers such as Isabelle with its Isar language allow for a focus on propositions themselves as is traditional in mathematics[3]. This fact, coupled with a language of tactics for performing the required reduction, expansions, and obvious equalities significantly reduce the burden on the programmer. Note however that often all the information required to perform computation is in the proofs themselves, carrying around their indices. As an example, the difficulty in dealing with `x == y` lies in the fact that it carries so little information. Conor McBride notes that to make use of a boolean, we must know its provenance. Clearly `True` is meaningless unless we had remembered that it was generated from the test `x == y` which is precisely what the inspection pattern seeks to transport. A better alternative to mindlessly hand reducing `lookupVar` would be to change its implementation to make use of a type which internalizes the fact that some element has membership. Indeed the mechanization of type-safety already "cheats" by matching on the proof of well-formedness of the environment in order to restrict the head of the context and environments in `lookupSafe`. Instead of taking an external notion of well-formedness dependent types can perform evaluation directly on well-formed terms themselves. In particular terms can be indexed by the context and type in which they are well-defined. This is at the heart of the correct-by-construction style of programming discussed in the section.

2.5.4 Verification with dependent types

It is somewhat traditional in dependently typed languages to prove the safety of evaluation via a correct-by-construction expression language [Bra05a]; [AC99];[Chl13, pp. 30–38]. Re-expressing STLC terms in this way is almost a direct transcription of the original `WfStlc` type.

```
data Exp : Ctx -> Ty -> Type where
```

[3]The reader may note that this mechanization is only correct up to termination. There are a wealth of strategies to draw upon for proving termination include PHOAS encoded closures [Chl08]; [LH09], appealing to accessibility proofs as in [Swi12], reformalizing with a small-step semantics, or of course using a timeout parameter as in CompCert.

```
Lit : (n : Nat) -> Exp ctx TyNat
Var : Elem t ctx -> Exp ctx t
Add : Exp ctx TyNat -> Exp ctx TyNat
   -> Exp ctx TyNat
Lam : (t1 : Ty)
   -> Exp (t1 :: ctx) t2
   -> Exp ctx (t1 `TyArrow` t2)
App : Exp ctx (t1 `TyArrow` t2)
   -> Exp ctx t1
   -> Exp ctx t2
```

This is because the certificate of well-formedness always had access to its indices—the `Stlc` expressions themselves—and therefore carried all the computational information of the raw `Stlc` terms required by evaluation. The sole changes to `WfStlc` are to remove the raw `Stlc` terms, and to re-express variable placement in terms of context membership. This is part of the desire to shift away from a focus on propositions, and describing how incidentally correct definitions reduce subject to external constraints. Instead because of the intrinsic focus on the shapes of proofs it is advantageous to shift as much work as possible to this process of inspection. The proofs of context membership are exemplary. In `evalSafe` proving correct handling of variables first tries to reduce the string equality check against the top of the context y by `x == y`. This fact is then used to manually expand or contract the body of the goal as required. Instead this equality can be internalized by framing membership as a proof that

1. the head of a list is a member; and
2. if an element is a member of a tail, then it is an element of an extended list.

in Idris

```
data Elem : a -> List a -> Type where
  Here : (x : a) -> (xs : List a) -> Elem x (x :: xs)
  There : (y, x : a) -> (xs : List a) -> Elem x xs
       -> Elem x (y :: xs)
```

By matching on this proof something is learned about the top of the context. Namely that it is either the member x, i.e. that `x == y = True`, or that the top of the stack is some other value, and x can be found in the tail. Curiously names are no longer explicitly carried by binders. This is because the new encoding expresses variables using De Bruijn indices [Bru72] in disguise. De

Bruijn indices encode variables as a number indicating the number of new binders between the variable and its own binder. Alternatively De Bruijn indices can be thought of as indicating the position in the context that the variable occurs. A rigorous isomorphism between De Bruijn indices and context membership proofs is given on appendix E on page 208.

To understand how to evaluate type indexed terms it's useful to first consider how to evaluate raw `Stlc` terms. The `Eval` relation in listing 2.3 on page 44 can be rewritten as a function—although the new formulation fails termination checking

```
eval : Env -> Stlc -> Value
eval env (Lit n) = VLit n
eval env (Var x) = case lookupVar x env of
  Just v => v
eval env (Add e1 e2) = case (eval env e1, eval env e2) of
  (VLit n1, VLit n2) => VLit (n1 + n2)
eval env (Lam x t e) = VClo x t e env
eval env (App e1 e2) = case (eval env e1, eval env e2) of
  (VClo x t e1' env', v2) => eval ((x, v2) :: env') e1'
```

Augustsson [AC99] notes that the `Value` type in the above is really just a means of tagging values with their types to ensure that the evaluated program does not misuse a value. However this is unnecessary if the program has undergone type-checking because it is known a priori that there are no type errors. In a type-safe language where checking occurs at runtime such as Python the above program could more naturally be written as

Listing 2.5: Type respecting evaluation

```
eval env (Lit n) = n
eval env (Var elt) = lookupVar elt env
eval env (Add e1 e2) = eval env e1 + eval env e2
eval env (Lam x t e) = \v => eval ((x, v) :: env) e
eval env (App e1 e2) = eval env e1 (eval env e2)
```

Since the program is known to be type correct, values can be returned and used without the need for tagging. The type of closures is particularly simple because there's no need to explicitly capture the environment when the host closure already captures `x`, `env`, and `e`. Defining the above definition of evaluation is relatively trivial using the correct-by-construction approach because the return type can be *calculated*

```
eval : Exp ctx t -> ?a
```

What is the type of the hole `?a` in the above? The type information about the input expression is locally available via `t`. Reiterating the act of constructing an expression at `t` guarantees it passes type checking when ascribed with the type `t`. The resulting type is precisely the types of the values captured by tagged `Values`.

```
EvalTy : Ty -> Type
EvalTy TyNat  = Nat
EvalTy (TyArrow t1 t2) = EvalTy t1 -> EvalTy t2
```

Remarkably the `eval` function in listing 2.5 will nearly pass typechecking, the only remaining problems relate to variables.

```
eval : ?env_ty -> Exp ctx t -> EvalTy t
eval env (Var elt) = lookupVar elt env
...
```

The current definition of environments map variables to explicitly tagged `Values`. Fortunately variables themselves, being De Bruijn encoded, carry proof of their type. A context membership proof is simply a statement of the position in the context at which the variable's type can be found. Thus the environment *must* follow the shape of the context which was precisely the structure of well-formed environments on figure 2.5 on page 44. Moreover since values no longer need tagging, rather than carrying a proof of the tagged value's well-formedness, the raw value of the correct type can be placed directly in the environment

```
data Env : Ctx -> Type where
  Nil : Env []
  Cons : (t : Ty) -> (ctx : List Ty)
      -> (value : EvalTy t) -> Env ctx
      -> Env (t :: ctx)
```

Then to perform variable lookup the membership proof is treated as index into the environment

```
lookupVar : Elem t ctx -> Env ctx -> EvalTy t
lookupVar (Here t ctx)         (Cons t ctx v env)  = v
lookupVar (There t' t ctx elt) (Cons t' ctx v env) =
  lookupVar elt env
```

Note that by matching on the proof of membership we learn that the environment is non-empty. Additionally matching on `Here` restricts the value `t : Ty` under the cons, and thus we learn that the type of v must be `EvalTy t`, the required type of the variable.

Thus the final type of the sound evaluation procedure can be given

```
eval : Env ctx -> Exp ctx t -> EvalTy t
```

This style of variable binding is both intuitive and flexible. Unfortunately it comes at a non-zero cost since the correctness of any variable manipulation must be immediately proven rather than performing a sequence of "incidentally correct" transformations and proving their correctness at the end. While strategies exist for avoiding this overhead, in this setting the extra proofs required are an acceptable overhead. The next section presents a brief coverage of strategies for representing binders in way that allows for handling them in a provably correct way.

2.6 A brief history of binding strategies

The problem of binding and representation of variables is an important and well-studied topic in the area of language development and verification [Ayd+08]; [Chl08]. Strategies include the usage of host binders as in (parametric) higher order abstract syntax (HOAS and PHOAS) [KJ12]; [ALY09]; [PE88]; [Chl08]; [LH09]; [PS99], locally nameless representations mixing De Bruijn indices with concrete names distinguishing bound and free variables [Cha12]; [MM04a], the globally unique "Barendregt convention", tools for code generation [Sew+07]; [Her05], and a wealth of patterns and libraries [Kme15]; [WYS11]; [BP99]. Indeed, within the development of $\mathscr{L}$ proofs, performing various forms of variable and context accounting are representative of almost a quarter of the total code.

2.6.1 Parametric Higher Order Abstract Syntax

In Higher Order Abstract Syntax (HOAS) variables in the object language are directly represented by variables in the host language. For example the λ-calculus from listing 2.2 on page 42 is captured in Idris using HOAS by the following definition.

```
data Exp : Ty -> Type where
  Lit : Nat -> Lam TyNat
  Add : Exp TyNat -> Exp TyNat -> Exp TyNat
  Lam : (Exp t1 -> Exp t2) -> Exp (t1 `TyArrow` t2)
  App : Exp (t1 `TyArrow` t2) -> Exp t1 -> Exp t2
```

This definition exemplifies two important properties: there is no way to explicitly tag variables; and lambda abstractions are expressed as a real Idris

function. Given that any variable will be a well-formed `Exp` type, there's no need for a variable constructor because any variable can be placed directly in terms. For instance an addition function on two variables

```
add : Exp (TyNat `TyArrow` (TyNat `TyArrow` TyNat))
add = Lam (\x => Lam (\y => x `Add` y))
```

Unfortunately this type is not strictly positive so Idris disallows its use in total functions. Additionally the `add` definition is free to unpack, inspect, and manipulate its arguments, operating more like a macro than a real function.

Washburn and Weirich [WW03] observed that a clever application of parametricity stops functions such as `add` from inspecting their variables, dubbing the strategy "parametric higher order abstract syntax" or PHOAS. Moreover the strategy is strictly positive and thus admissible in the more stringent settings in e.g. Idris, Agda, and Coq.

In PHOAS object language abstractions are still encoded using host abstractions, however the type of the source variable is an abstract parameter v. Thus by parametricity, definitions such as `add` may only shunt variables around. Note that since variables are of abstract type, an explicit constructor for variable placement must be reintroduced (thus strictly speaking, PHOAS is not technically a HOAS). An untyped λ-calculus would be encoded with PHOAS by

```
data PExp : (var : Type) -> Type where
  Var : var -> PExp var
  Lam : (var -> PExp var) -> PExp var
  App : PExp var -> PExp var -> PExp var
```

Thus functions in type `(var : Type) -> PExp var` know nothing about the type `var` and therefore are unable to perform macro-like syntactic inspections.

Internalized type safety is readily recovered in PHOAS. Since variables are typed in this setting, the `var` parameter is simply indexed by its type in the usual way.

```
data PExp : (var : Ty -> Type) -> Ty -> Type where
  Var : var t -> PExp t
  Lit : Nat -> PExp TyNat
  Add : PExp TyNat -> PExp TyNat -> PExp TyNat
  Lam : (var t1 -> PExp t2) -> PExp (t1 `TyArrow` t2)
  App : PExp (t1 `TyArrow` t2) -> PExp t1 -> PExp t2
```

Unfortunately PHOAS has a few undesirable properties. In particular care must be taken if evaluation is to be mixed with syntax since evaluation inherently requires some knowledge about the shape of `var`. Thus corners which want to perform evaluation must ensure that the term is only inspected during evaluation, but additionally that it *not even be constructed* in other contexts. While such problems are not insurmountable, Idris is reluctant to perform reduction underneath binders thus complicating proofs about functions in EDSLs. Given the above problems, our development proceeds using a notion of name indexed De Bruijn indices to encode variables which will now be detailed.

2.6.2 Name indexed De Bruijn indices

De Bruijn indices are an elegant and simple means of representing variables, encoding variables as numbers representing the distance from their binding site. For example the term $\lambda x.\lambda y.x + y$ would be represented as $\lambda.\lambda.1 + 0$ since there is one binder—namely y—between x and its binding site. As an additional benefit, De Bruijn indices naturally grant a means of testing α equivalence, since variable names are unique. This simplicity granted by De Bruijn indices are traded for additionally complexity in substitution. For example naïvely reducing $(\lambda.\lambda.0 + 1)\ 0$ yields the obviously incorrect term $\lambda.0 + 0$. Thus when moving under binders, care must be taken to properly *shift* the value of the substituted term—in this case by adding 1 to variable 0. In a dependently typed setting it's possible to ask the compiler to check the correctness of all such substitutions by considering a hybrid encoding indexing De Bruijn indices by their names. This will also clarify why values must be shifted by a particular amount. A well-typed, name indexed De Bruijn variable is a proof of the name's membership in a context. Consider again the tired example of the typed λ-calculus

```
Ctx : Type
Ctx = List (String, Ty)
data DExp : Ctx -> Ty -> Type
  Var : Elem (x, t) ctx -> DExp ctx t
  Lam : (x : Name) -> (t : Ty) -> DExp ((x, t) :: ctx) t
     -> DExp ctx t
  App : DExp ctx (t1 `TyArrow` t2) -> DExp ctx t1
     -> DExp ctx t2
  ...
```

Thus an abstraction ascribes both a name and type to its variables, with the body defined in a scope extended by the newly bound variable. Additionally, Idris now rejects incorrect reduction as ill-typed. This is because variables are not simple numbers, they are numbers decorated with their *meaning*—a proof of where the variable name and type can be found in the context.

Consider again the improper reduction of $\lambda.(\lambda.\lambda.0 + 1)\ 0$. Suppose 0 and 1 are decorated with the names x and y. The inner scope $0 + 1$ has type `("x", TyNat) :: ("y", TyNat) :: ctx`. Supposing the outer 0 bound a variable with name z it has a reduced scope indicated by the type `ctx = [("z", TyNat)]`. Naïvely attempting to place the outer variable 0 produces a type error since the proof that z is found at index 0 no longer holds. More variables have been added and therefore the proof must be shown to hold in a *weakened* context i.e. z can still be found in a context with more names added. Clearly given $z \in \Gamma$ it is the case that $z \in x :: y :: \Gamma$. Making this rigorous transforms membership proofs in the same way that naked De Bruijn indices are transformed: point past the "added stuff" e.g. in Idris weakening a member of context Γ by Γ'

Listing 2.6: Context membership weakening

```
weaken : Elem x ctx -> (ctx' : List a) -> Elem x (ctx' ++ ctx)
weaken elt []           = elt
weaken elt (y :: ctx') = There (weaken elt ctx')
```

Thus as each additional variable y is added, x can be found one index further in the new context. In the case no more variables are to be added i.e. `ctx'=[]` the original membership proof still holds since x is found at the same location it was at originally.

Another important case to consider is the generalization of weakening known as *thinning*. Thinning is the insertion of additional information Γ' in the middle of a split context $\Gamma = \Gamma_1 +\!\!+ \Gamma_0$. This is necessary when for example performing weakening underneath a binder. Thinning is performed by case analysis on the split context Γ, divining the side—Γ_1 or Γ_0—in which a variable resides

1. When $x \in \Gamma_1$, no work is required to show that $x \in \Gamma_1 +\!\!+ \Gamma' +\!\!+ \Gamma_0$ since its index does not change.

2. When $x \in \Gamma_0$ it should be weakened by Γ' in the usual way.

In Idris the type system directs how to implement the transformation correctly by simultaneously unpacking the membership proof and the context.

```
thin : Elem x (ctx1 ++ ctx0) -> (ctx' : List a)
    -> Elem x (ctx1 ++ ctx' ++ ctx0)
```

First the case $x \in \Gamma_1$ is handled

```
thin {ctx1=x::ctx1'} Here         ctx' = Here
thin {ctx1=y::ctx1'} (There elt) ctx' =
  There (thin {ctx1=ctx1'} elt ctx')
```

Once `ctx1` is exhausted and x has not yet been found, the membership proof must be updated to look past the newly added elements in `ctx'`.

```
thin {ctx1=[]} elt ctx' = weaken elt ctx'
```

Encoding variables in this way not only ensures their correct handling but additionally allows for a highly liberal notion of a "name". In fact, arbitrary values can be tracked in a context. Incidentally this provides a convenient means of *reifying* arbitrary host expressions into a first order syntax that can be manipulated. This forms the basis of *proof by reflection*—the topic of discussed in the next section—a powerful method of generic theorem proving entirely within the language of dependent types.

2.7 Proof by reflection

One of the many advantages EDSL's offer is the usage of advanced host features to program with. Thus far while the host has been leveraged to prove the correctness of a variety of functions in $\mathscr{L}$, the work has been entirely manual. Most theorem provers provide a means of semi-automated theorem proving to simplify the process of proving classes of similar or repetitive problems.

The theorem prover Isabelle which distinguishes between proving and programming exposes a collection of *tactics* [Wen+17b]. Tactics can be implemented in Ocaml [Wen+17a] and attempt to solve obligations by fully-automated repeated goal inspection and refinement. Coq similarly employs the use of tactics, however it additionally is equipped with the Ltac [Del00] language, allowing the programmer to match on the current goal and current assumptions in order to try and automatically solve or simplify the proof obligation by producing a proof term in the underlying theory. Both Isabelle and Coq have been successfully used to produce large scale verified pieces of software with an impressive level of automation. In Isabelle this includes

the seL4 project [Kle+09], a verified micro-kernel and in Coq the proof of the Four-Color theorem [Gon07].

The proof of the four-color theorem makes extensive use of a theorem proving strategy unique to dependent types known as *proof by reflection*. In proof by reflection propositions are reified into some first order syntax, an inductive type that can be traversed in the usual language programming paradigms. These syntactic terms can be analyzed, reduced and then *reflected* back into their original form. Thus the act of searching for a proof of a proposition is done entirely within a single language. A key advantage provided by access to a first order representation of some proposition identified by van der Walt and Swierstra [WS12] is that the values of inputs can be enumerated. This strategy is exemplified by proving properties about boolean conjunctions and disjunctions in some set of variables. Usually this would require 2^n pattern matches on the input variables to test that all possible sets produce a true statement, however by analyzing the variables in syntax these values can be automatically instantiated with all possible values and hence the proposition can be tested in all possible environment.

The use of a single language to perform theorem proving is both elegant and intuitive. In fact, two examples of proof by reflection have already been given. The first in section 2.2.1 on page 28 where a number is proved to be even or not by the decision procedure `isEven` given in listing 2.1 on page 29. The second was of course proof of a De Bruijn index being well-formed. As van der Walt and Chlipala note, proof by reflection is made considerably more useful when arbitrary expressions can be injected into syntax trees without the need for an explicitly reified representation. This allows for instance the construction of highly general monoid equality solver where the "variables" become arbitrary expressions.

2.7.1 Proving equality of monoid expressions

A monoid is a set A equipped with an associative binary "append" operator $\cdot \times \cdot$ and an identity or "nil" element ϵ satisfying the following identify laws

$$\forall a \in A.\epsilon \times a = a$$
$$\forall a \in A.a \times \epsilon = a$$

The equation for the associativity of append is of course given by

$$\forall a_1\ a_2\ a_3 \in A.a_1 \times (a_2 \times a_3) = (a_1 \times a_2) \times a_3$$

Thus two equations in a monoid are equal if the parentheses in the left and right hand sides can be "shuffled" about to yield character for character identical expressions. Thus to test if two equations are equal programmatically, we must identify an equal parenthesization. As seen with System F_ω the usual approach to such a problem is to repeatedly rewrite the equations using some set of *confluent* rules to arrive at a canonical normal form. The confluence property of a rewrite system simply ensures that the order in which reductions are applied has no bearing on the normalized value. Since there can only be one such form, and the rewrite procedure is strongly normalizing equivalent terms are guaranteed to normalize to identical forms. In the case of a monoid it can reasonably be seen that the *right association* of all parenthesis is unique, forming the basis of our formalization. Intuitively this reduces expressions to strings of variables drawn from A—strings of course can be equality checked under the usual notion of lexical equality.

In Idris a well-formed monoid expression in some quantified variables tracked by a context is given by

```
data SimpleMonoidExp : List String -> Type where
  MUnit : SimpleMonoidExp ctx
  App : SimpleMonoidExp ctx -> SimpleMonoidExp ctx
     -> SimpleMonoidExp ctx
  Var : Elem x ctx -> MonoidExp ctx
```

Thus the equation $(a_1 \times a_2) \times a_3$ in Idris is written

```
(Var Here `App` Var (There Here)) `App`
 Var (There (There Here))
```

where the type explicitly states the order in which the free variables were introduced `SimpleMonoidExp ["a1", "a2", "a3"]`. The type of normal forms is also easily expressed by restricting the left-hand-side of the `App` constructor to values. Additionally since values are always placed to the left of an append operation, there's no need for an explicit `Var` constructor.

```
data NfSimpleMonoidExp : List String -> Type where
  NfUnit : NfSimpleMonoidExp ctx
  NfApp : Elem x ctx -> NfSimpleMonoidExp ctx
       -> NfSimpleMonoidExp ctx
```

Thus the normalized version of the expression $(a_1 \times a_2) \times a_3$ is

```
Var Here `NfApp`
(Var (There Here) `NfApp`
   (Var (There (There Here)) `NfApp` NfUnit))
```

Clearly terms in `NfSimpleMonoidExp` can be checked for equality by first matching constructors and then testing element equality in the case of `NfApp`.

Given a sound translation from `SimpleMonoidExp` to the normalized type `NfSimpleMonoidExp`, this construction can be used to automatically test the equality of terms from `SimpleMonoidExp`. Incidentally this strategy can be extended to automate equality proofs of arbitrary monoid expressions in a reflective style. The extensions employs two key insights. The first is the standard reflective trick of indexing the syntactic monoid expression by the underlying semantic value whose equality is to be tested. The second insight is to consider variables to be arbitrary expressions rather than static strings. Of course equality checking of arbitrary terms is undecidable, fortunately the encoding of variables themselves can be used to test for equality. This is a consequence of reflexivity, since if two variables are equal, they point to the same value in the context whose equality follows from reflexivity. The necessary equivalence relation on variables offers clarity

```
data ElemEq : Elem x xs -> Elem y xs -> Type where
  HereHere : ElemEq {y=x} {xs=x::xs'}
                    (Here {x=x}) (Here {x=x})
  ThereThere : ElemEq elt elt'
            -> ElemEq (There elt) (There elt')
```

Thus two variables pointing to the top of the context are equal, requiring that `x = y` because `Here` restricts the element to be the same as the top of the stack i.e. `Elem x (x :: xs)`. Additionally two variables pointing to some other equal point of the context are also equal. When the final piece of the equality proof `HereHere` is unwrapped we discover that it is only ever the case that `x` and `y` were equal as formalized by the following proof

```
elemEqId : {elt : Elem x xs} -> {elt' : Elem y xs}
        -> ElemEq elt elt' -> x = y
elemEqId HereHere = Refl
elemEqId (ThereThere eltEelt') = elemEqId eltEelt'
```

Extending monoids to abstract expressions

Having recovered a means of automatically identifying variables, translating the `SimpleMonoidExp` type to encode arbitrary expressions is straightforward[4]

[4]Note that this definition is parametric in some explicitly defined monoid on `a` given by `ExplicitMonoid`, a record exposing the unit and append operations associated with

```
data MonoidExp : ExplicitMonoid a -> List a -> a
              -> Type where
  MUnit : MonoidExp m ctx ExplicitMonoid.unit
  App : MonoidExp m ctx x -> MonoidExp m ctx y
     -> MonoidExp m ctx (x `append` y)
  Var : Elem x ctx -> MonoidExp m ctx x
```

Following the standard reflective approach the unit constructor `MUnit` is indexed by the true underlying semantic unit of the monoid. Similarly the `App` constructor is indexed by the appending of some semantic `x` and `y` taking as arguments their reified sub-trees. Finally a variable `x`—semantically an arbitrary expression—is placed by giving a proof of its membership in the context. Moreover this proof is indexed by the semantic expression in question which is placed as the index of the overall term. Ultimately such expressions should be provably equal to one another through a reflective "tactic" e.g.

```
eqMonoidExp : MonoidExp m ctx x -> MonoidExp m ctx y
           -> Maybe (x = y)
```

As an example of how to manually reify some expression, consider addition with unit 0. The expression $(x, y) \mapsto x + 2 + y$ can be transformed into a syntax tree as follows

```
x2y : (x, y : Nat)
   -> MonoidExp PlusMonoid [x, 2, y] ((x + 2) + y)
x2y x y = (Var Here `App` Var (There Here)) `App`
           Var (There (There Here))
```

Note that both the constant 2 and the variables x, y had to be placed as variables since monoids can only express trees of variables and unit values appended to one another. The translation of normal forms is similarly trivial

```
data NfMonoidExp : ExplicitMonoid a -> List a -> a
                -> Type where
  NfUnit : NfMonoidExp m ctx ExplicitMonoid.unit
  NfApp : Elem x ctx -> NfMonoidExp m ctx y
       -> NfMonoidExp m ctx (x `append` y)
```

This is almost identical to the `NfSimpleMonoidExp` except the context now tracks a set of expressions rather than variable names. Particularly notable is that a translation from an `x` indexed `MonoidExp` to an `x` indexed `NfMonoidExp` is sound if their indices match. Intuitively given that the index represents

---

the monoid.

an abstract interpretation of the syntax tree, identification of the indices implies semantic equivalence of the two monoid expressions. This notion is given more precisely and is formalized in chapter 3 on page 66 lying at the heart of $\mathscr{L}$.

The next step required to prove the equality of two monoid expressions is to perform sound normalization.

Normalizing monoid expressions

Having access to a strongly typed description of a normalized expression makes the task of normalization relatively simple. Following the strategy of Brady [Bra13a] and Slama [SB17] the soundness of the normalization procedure is given *by construction*. Hence in order to normalize some monoid expression that reifies x, at each step a normalized expression is produced in some new index x' along with a proof that it's equal to the input x i.e. $x = x'$.

```
nfMonoidExp : MonoidExp m ctx x
           -> (x' : a ** (NfMonoidExp m ctx x', x = x'))
```

Thus the above returns not only a syntactic description of the normalized expression, but additionally a proof $x = x'$ of how to transport the expression x to its normal form x'. Concretely this proof will be some repeated application of the substitution rule using append's proof of associativity.

The normalization procedure itself is simple. Intuitively an append expression $l \times r$ can be normalized by normalizing the left-hand side l and affixing the normalized right-hand side to its base. Thus normalization takes as arguments both an expression to normalize e, and a normalized spine n to affix at the base of the normalized e. An example of how to normalize $(a_1 \times a_2) \times a_3$ given a normalized spine argument of n_0 is given in figure 2.6.

```
nfMonoidExp' : MonoidExp m ctx x
            -> (y : a) -> NfMonoidExp m ctx y
            -> (z : a ** (NfMonoidExp m ctx z,
                          x `append` y = z))
nfMonoidExp' (App {x=x1} {y=x2} e1 e2) y yNf =
  let (x2y ** (e2YNf, eq1)) = nfMonoidExp' e2 y yNf
      (x1x2y ** (e1e2YNf, eq2)) = nfMonoidExp' e1 x2y e2YNf
  in (x1x2y **
      (e1e2YNf,
       replace {P=\v => v = x1x2y}
               (appendAssociative x1 x2 y)
```

```
                (replace {P=\v => x1 `append` v = x1x2y}
                         (sym eq1) eq2)))
```

As is the case in `nfMonoidExp`, `nfMonoidExp'` a normalized expression coupled with a proof that its semantics—the index z—is equivalent to x i.e. the index of the input. The implementation clarifies the fact that the normalization procedure calculates not only a normal expression but a proof transporting x to z. This proof is a description of how to perform normalization on the *reflection* x of the reified expression. In this case the proposition to prove is

```
(x1 `append` x2) `append` y = x1x2y
```

given the facts that

```
eq1 : x2 `append` y = x2y
eq2 : x1 `append` x2y = x1x2y
```

drawn from the recursive calls. This follows trivially by substituting for `x2y` and then applying the associativity rule. These two operations are performed by the calls to `replace`.

Normalizing unit and variables are trivial: in the case of unit simply return the right spine; and in the case of variables produce a new append node with the variable on the left and the normal spine on the right.

```
nfMonoidExp' MUnit y yNf = (y ** (yNf, unitLeft y))
nfMonoidExp' (Var {x=x} elt) y yNf =
  (x `append` y ** (elt `NfApp` yNf, Refl))
```

These proofs of index equality are also trivial. In the case of unit it must be shown that `unit `append` y = y` which follows from the unit law `unitLeft y`. Finally the case of appending a variable to a spine is already in the proper from, requiring no work and thus returning `Refl`.

While the procedure `nfMonoidExp'` is correct, a "hole-filling" spine argument is always required. The top-level function `nfMonoidExp` selects a suitable default value of `unit`.

```
nfMonoidExp : MonoidExp m ctx x
           -> (x' : a ** (NfMonoidExp m ctx x', x = x'))
nfMonoidExp {x=x} e1 =
  let (x' ** (e1Nf, xEx')) = nfMonoidExp' e1 unit NfUnit
  in (x' ** (e1Nf, replace {P=\v => v = x'}
                           (unitRight _)
                           xEx'))
```

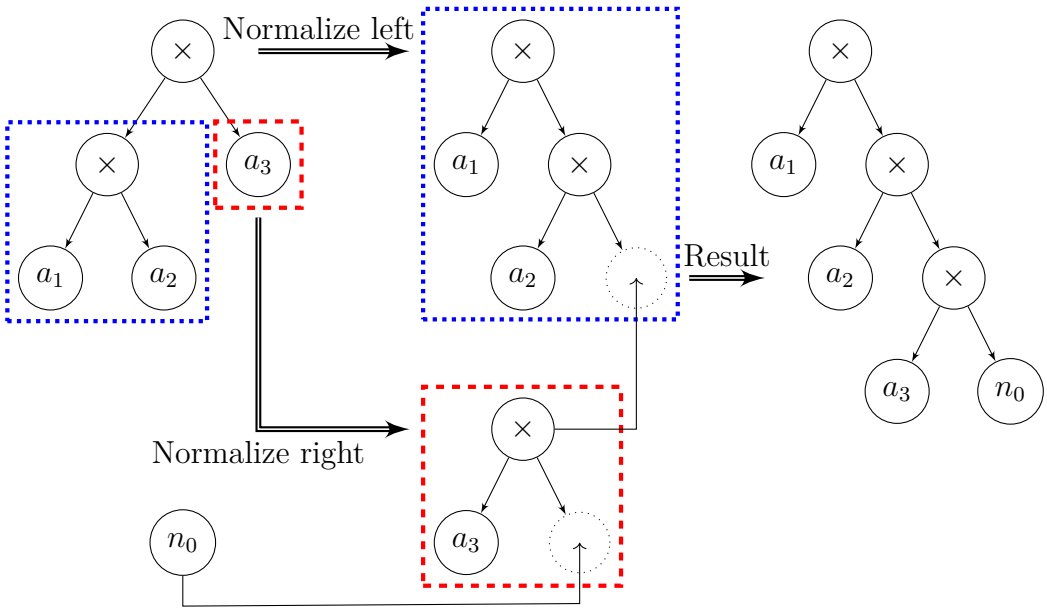


Figure 2.6: An example of the normalization procedure.

As usual a proof to transport the new index `x `append` unit` to the input index `x` is required which follows precisely from unit laws.

Given a sound means of normalizing reified monoid expressions, it's possible to provide a sort of tactic for automatically solving monoid equations.

Automatically proving monoid equalities

Formally the problem of deciding if two monoid expressions are equal is

```
eqMonoidExp : {a : Type} -> {x, y : a}
           -> MonoidExp m ctx x -> MonoidExp m ctx y
           -> Maybe (x = y)
```

Note that this procedure is sound but not *complete*. Completeness is tricky because our monoid variables represent real values of type `a`. Thus there will be instances where two variables point to distinct locations in the context and yet the values at those locations are equal. In these situations the equality of `x` and `y` cannot be given, nor of course can the equality be disproven since nothing is *known* about the values. One solution is then to give a complete procedure which calculates a set of requirements—constraints on variable

values—which must be proven in order to come to a conclusion. While the idea won't be deployed here, it's crucial to the development of $\mathscr{L}$.

The function `eqMonoidExp` becomes trivial given a test for normalized monoid expressions

```
eqNfMonoidExp : NfMonoidExp m ctx xs
             -> NfMonoidExp m ctx ys
             -> Maybe (xs = ys)
```

Intuitively this should hold because the indices of `NfMonoidExp` are *almost* unique and thus the equality check on the syntactic reification should *almost* imply equality on the indices. As earlier remarked, the only point where an expression's index is not uniquely determined by the syntax is at variables which could be equal to anything in the absence of information. Thus the proof follows by simultaneous case analysis on the arguments discriminating between append and unit, identifying unit with itself, and identifying two applications when their tails can be identified and the left-hand variables point to the same location in the context.

```
eqNfMonoidExp NfUnit NfUnit = Just Refl
eqNfMonoidExp (NfApp x1Elt x2Nf) NfUnit = Nothing
eqNfMonoidExp NfUnit (NfApp y1Elt y2Nf) = Nothing
eqNfMonoidExp (NfApp x1Elt x2Nf) (NfApp y1Elt y2Nf)
  with (elemEq x1Elt y1Elt, eqNfMonoidExp x2Nf y2Nf)
  eqNfMonoidExp (NfApp x1Elt x2Nf) (NfApp y1Elt y2Nf)
    | (Yes eltEq, Just eq) with (elemEqId eltEq)
    eqNfMonoidExp (NfApp x1Elt x2Nf) (NfApp y1Elt y2Nf)
      | (Yes eltEq, Just Refl) | Refl = Just Refl
  eqNfMonoidExp (NfApp x1Elt x2Nf) (NfApp y1Elt y2Nf)
    | (Yes eltEq, Nothing) = Nothing
  eqNfMonoidExp (NfApp x1Elt x2Nf) (NfApp y1Elt y2Nf)
    | (No eltEq, nfeq) = Nothing
```

From `eqNfMonoidExp` proof of `eqMonoidExp` follows directly. Normalize the inputs, test their equality, and if it can be shown, then apply the resulting equality proofs to transport the indices to their original values. The full proof is given in appendix B.2 on page 192

While the function `eqMonoidExp` can be used to prove the equality of some terms, because the syntactic reifications of the terms to test must be given by hand, it doesn't feel quite automatic. Languages lacking a distinct tactics language such as Idris[5] and Agda solve this problem by providing

[5]Technically Idris did have a tactics language, however it's been superseded by its first-class reflection mechanisms.

first-class reflection support.

2.7.2 First-class reflection

Recently Agda has been extended to provide an API for reflecting the goal [Wal12] into a first order syntax tree that can be manipulated in the standard language similar for example to Template Haskell [SJ02] or C++ templates. This solves the problem of constructing the "monoid view" of a term because it can be constructed by reflecting terms into first order syntax and then building a `MonoidExp` by traversing this tree.

Similarly Idris supports a first-class reflection mechanism allowing for functions to perform pattern matching on syntax. More recently Idris has been extended with the more powerful *elaborator reflection* [Chr14]; [CB16] a means of exposing the compiler's hole solving machinery—among other things—"as a primitive monad in Idris itself", as Christiansen puts it, to perform metaprogramming. Together with first-class reflection, elaborator reflection exposes a powerful means of automatically solving proof obligations.

2.8 Summary

In this chapter a series of progressively more expressive type-systems have been presented, covering the core set of type-level features which are used in the design and development of the language $\mathscr{L}$ explored in the next chapter. In particular system F_ω and synthetic dependent types both expose powerful means of statically reasoning about terms via a distinct language of types. We have shown how carefully crafted static types can restrict the values of runtime terms by considering types as computations over *kinds*.

This chapter has also given a brief introduction to the usage of dependent types in programming and theorem proving. Several patterns for theorem proving were provided by the ruling out of invalid terms a priori. This is the correct-by-construction pattern. We have seen how to use dependent types to embed languages with weaker type systems such as the STLC in addition to strategies for representing variable binders with (P)HOAS style using host binders to encode object language binders. However due to the drawbacks of PHOAS a correct-by-construction notion of De Bruijn indices is proposed using proofs of membership within an explicitly tracked context.

Finally this chapter covered *proof-by-reflection* a powerful means of automated theorem proving unique to dependent types. In chapter 4 on page 108 we will see how this technique can be used to interface with EDSLs.

Chapter 3

Verified typing with explicit coercions

This chapter will work to develop a language $\mathscr{L}$ meant to be implemented within existing theorem provers. A means of ascribing terms with their semantics will be described. Following the in the spirit of Dependent ML [Xi07] the language of term semantics will often be referred to as the language of index expressions. Additionally a method of statically reasoning about these index expressions within the host will be presented. This type-system will take the effective shape of a limited form of synthetic dependent types [CW00]. However notably the system presented will not use an algorithmic set of typing rules, instead relying on the programmer to apply explicit coercions as required. Together the above criteria make available an ideal playground for exploring how dependent types can aid in writing correct-by-construction programs and transformations where the correctness proofs arise from the semantics captured by terms' indices.

The development is broken down into three primary goals:

1. Develop an embeddable language of statics: the term *index expressions*, or semantics, along with a runtime language and its dynamics: the *semantics indexed expressions*.

2. Present a method of explicit coercion using only their semantics, suitable for embedding in existing theorem provers.

3. Show how the strategy can be used in Idris to give a correct by construction implementation of a language with an explicit conversion

rule.

In particular $\mathscr{L}$ will have a simple, constructively verified core which can be built upon using e.g. the deep-shallow embedding of Svenningsson and Axelsson [SA15] in chapter 4 on page 108.

As language features are refined, important properties will be given in the traditional relational style and compared to the correct by construction EDSL.

3.1 A language of term semantics

One of the primary goals of $\mathscr{L}$ is to ensure the safety of program transformations using dependent types. Every term in $\mathscr{L}$ is indexed by an index expression, capturing its semantics, and thus the safety of a transformation will follow when this index is respected. Having access to a term's semantics also allows for asserting and proving correctness properties directly on this index expression directly in the host.

Motivating the design of the language $\mathscr{L}$ are several important considerations:

1. All programs carry their semantics as an index, given by an index expression.

2. Case expressions should allow for the introduction of knowledge by refining the scrutinee type.

3. Facts collected about index expressions, i.e. about a program's semantics, should provide utility e.g. to eliminate unreachable code.

4. The developer should be able to prove properties of programs without relying on an opaque solver such as an external SMT solver which exists outside the system and thus is not easily directed by the programmer.

We start by first defining the language of simple expressions $\mathscr{L}$, presented in figure 3.1, and then proceed by exploring the index language. This is in order to have a directed design of the index language—i.e. the language of expression semantics.
The language includes support for variables; peano numbers; and lists; in addition to case analysis on lists; and fold functions, or eliminators for

Expressions
$$
\begin{aligned}
e ::= {} & v \in \mathbb{N} \\
& \mid e + 1 \mid 0 \\
& \mid [] \mid e :: e \\
& \mid \text{caselist } e\ ((x :: xs) \Rightarrow e); ([] \Rightarrow e) \\
& \mid \text{elimlist } e\ (x, acc \mapsto e)\ e \\
& \mid \text{elimnat } e\ (x, acc \mapsto e)\ e
\end{aligned}
$$

Figure 3.1: Syntax of $\mathscr{L}$: semantics indexed expressions.

iterating over lists and natural numbers. Note that variables are encoded in De Bruijn style which will usually be written as v[1]. In chapter 4 on page 108 we show this set of language features is expressive enough to capture an interesting variety of familiar programs, or functions. Before giving a type-system for this language, we first explore the language index expression, or expression semantics.

In order to further motivate the language design of index expressions, consider the problem of the safe head—or tail—function without the use of static analysis or via the introduction of an external solver. Clearly this requires both a notion of lists capturing the semantics of the input list whose head is to be projected. It is also apparent that natural numbers should be expressible in order to capture the value of the projected head element. Thus at a minimum the index expressions must be defined as follows

$$
\begin{aligned}
S ::= {} & w \in \mathbb{N} \\
& \mid S + 1 \mid 0 \\
& \mid [] \mid S :: S
\end{aligned}
$$

Note that similar to the semantics indexed expressions, variables w are De Bruijn encoded. Next, consider safe head using full-spectrum dependent types which is a straight-forward task. First sized lists—called `Vect`—are given by constructing a new type that internalizes the proof of length.

Listing 3.1: Size indexed lists

```
data Vect : Nat -> Type -> Type where
```

[1]The meta-variable v is also used for values, however the context will always make clear its meaning.

```
Nil : Vect Z a
(::) : a -> Vect n a -> Vect (S n) a
```

Thus a `Vect` is either a `Nil` zero length empty list, or a `(::)` (cons) of length `n+1` given an element and some `n` sized list. By exposing the size of a term in its type, restrictions on the shape of the list can be made. For example, the total *head* function requires a non-zero size input

```
safeHead : Vect a (S n) -> a
safeHead [] impossible
safeHead (x :: _) = x
```

To reiterate, the `Nil` case has length zero, and there is no substitution in `n` that makes `Z` equal to `S n`. This fact means the first case is unreachable. Note `safeHead` could alternatively be given on naked lists with an external proof of length e.g.

```
safeHead' : (xs : List a) -> (length xs = Z -> Void) -> a
```

However the `Nil` case here requires an explicit proof of why the length is non-zero. There is a close relationship between internal and external proofs explored in the context of typed and untyped λ-calculi by Reynolds [Rey03] and later by Hsiang-Shang Ko and Gibbons with dependent types [KG13].

The sophisticated elaboration [Bra13b] technique used by Idris coupled with dependent pattern matching [Coq92] makes such functions easy to write but leave the underlying mechanism opaque. Implementing the same function in Coq which lacks strong support for programming with heavily dependent types [Chl13, pp. 140–165] illuminates a means of dependent case analysis. A first attempt at the safe head function will fail in older versions of Coq, citing the below pattern match as being non-exhaustive.

```
Definition safeHd1 {A : Set} {n : nat} (xs : vect A n) : A :=
  match xs with
    | Cons x xs' ⇒ x
  end.
```

When discovering that `xs` is `Cons`, we learn that the length `n` is non-zero, however this information is not pushed into the type system. Instead a `safeHd` function must defer to an explicit auxiliary function which *is* exhaustive. The return type is given as a function of `n`, so that no `A` typed value is required when `n` is zero.

```
Definition safeHd' {A : Set} (n : nat) (xs : vect A n) :=
  match xs in vect _ n return match n with
```

Index expressions
$$
\begin{aligned}
S ::= {} & w \in \mathbb{N} \\
& \mid S + 1 \mid 0 \\
& \mid [] \mid S :: S \\
& \mid \mathit{caselist}\ S\ ((j :: js) \Rightarrow S);\ ([] \Rightarrow S) \\
& \mid \mathit{elimlist}\ S\ (j, \mathit{acc} \mapsto S)\ S \\
& \mid \mathit{elimnat}\ S\ (j, \mathit{acc} \mapsto S)\ S
\end{aligned}
$$

Figure 3.2: Syntax of index expressions: program semantics

```
                | O ⇒ unit
                | S m ⇒ A
              end with
  | Nil ⇒ tt
  | Cons x xs' ⇒ x
end.

Definition safeHd {A : Set} {n : nat} (xs : vect A (S n)) : A :=
  safeHd' (S n) xs.
```

The return type of `safeHd'` is itself a match clause which only reduces when the vector length `n` is split, obviating the need for intricate pattern matching strategies such as that presented by Gundry and McBride [GM12] or Abel and Pientka [AP11]. Equipped with `safeHd'` the definition of `safeHd` becomes trivial—the shape of `n` is known to be non-zero, allowing for the return type of `safeHd'` to reduce to `A` as required [2].

Thus to support this pattern the language of index expressions will include a native case analysis construct for decomposing lists. In fact the language of index expressions will mirror *all* the syntactic constructs in $\mathscr{L}$ allowing us to capture the semantics of structural recursion on numbers and lists. The full syntax of index expressions is given in figure figure 3.2.

As noted a term's index expression is dependent upon its type. Since programs consist only of numbers and lists it is sufficient to restrict ourselves to these two types

$$\tau ::= \mathit{List} \mid \mathit{Nat} \tag{3.1}$$

[2]Note that recent version of Coq will infer the explicit `match in return` clause given above, allowing for the more direct `safeHd1` function.

Index expression $\quad \Delta ::= (j, \tau), \Delta \mid \emptyset$
Contexts

$\boxed{\Delta \ni_{\mathbb{N}} (\mathbf{W}, \tau)}$ (well-formed De Bruijn indices)

$$\frac{}{(x,\tau), \Delta \ni_0 (x,\tau)} \qquad \frac{\Delta \ni_w (x,\tau)}{(y,\tau), \Delta \ni_{w+1} (x,\tau)}$$

(a) Contexts of index expression variables.

$\boxed{\Delta \vdash i : \tau}$ (i is a well-formed index expression)

$$\text{SVar}\frac{\Delta \ni_w (x,\tau)}{\Delta \vdash w : \tau}$$

$$\text{SZero}\frac{}{\Delta \vdash 0 : \mathit{Nat}} \qquad \text{SSuc}\frac{\Delta \vdash i : \mathit{Nat}}{\Delta \vdash i + 1 : \mathit{Nat}}$$

$$\text{SNil}\frac{}{\Delta \vdash [] : \mathit{List}} \qquad \text{SCons}\frac{\Delta \vdash i : \mathit{Nat} \quad \Delta \vdash is : \mathit{List}\ \tau}{\Delta \vdash i :: is : \mathit{List}\ \tau}$$

$$\text{SCaselist}\frac{\Delta \vdash i_1 : \mathit{List} \quad (j, \mathit{Nat}), (js, \mathit{List}), \Delta \vdash i_2 : \tau \quad \Delta \vdash i_3 : \tau}{\Delta \vdash \mathit{caselist}\ i_1\ ((j :: js) \Rightarrow i_2)\ ; ([] \Rightarrow i_3) : \tau}$$

$$\text{SElimlist}\frac{\Delta \vdash i_1 : \mathit{List} \quad (j, \mathit{Nat}), (acc, \tau), \Delta \vdash i_2 : \tau_2 \quad \Delta \vdash i_3 : \tau_2}{\Delta \vdash \mathit{elimlist}\ i_1\ \ (j, acc \mapsto i_2)\ ; (i_3 : \tau_2)}$$

$$\text{SElimnat}\frac{\Delta \vdash i_1 : \mathit{Nat} \quad (x, \mathit{Nat}), (acc, \tau), \Delta \vdash i_2 : \tau_2 \quad \Delta \vdash i_3 : \tau_2}{\Delta \vdash \mathit{elimnat}\ i_1\ (x, acc \mapsto i_2)\ i_3 : \tau_2}$$

(b) Typing rules for index expressions.

Figure 3.3: Well-formedness of index expressions, i.e. expression semantics.

Thus an index expression is only well formed when it satisfies the arity-3 well-typing relation written $\cdot \vdash \cdot : \cdot$ in figure 3.3. The relation $\Delta \vdash i : \tau$ asserts the well-formedness of i stating it has type τ in the *context* Δ. A context Δ is a linear list, mapping *names in scope* to their types. Contexts are additionally equipped with a membership relation $\Delta \ni_w (i, \tau)$ given in figure 3.3a which capture the idea that a well-formed De Bruijn index always yields a mapping (i, τ) in the list Δ.

The translation of this language within Idris is an exercise in formatting

the typing rules from figure 3.3

```
IxCtx : Type
IxCtx = List (String, Ty)
data Ix : IxCtx -> Ty -> Type where
  S : Ix ictx Nat -> Ix ictx TyNat
  Z : Ix ictx TyNat
  Var : Elem (x, k) ictx -> Ix ictx k
  Nil : Ix ictx TyList
  (::) : Ix ictx TyNat -> Ix ictx TyList -> Ix ictx TyList
  CaseList : Ix ictx TyList
          -> (x, xs : Name)
          -> Ix ((x, TyNat) :: (xs, TyList) :: ictx) k
          -> Ix ictx k
          -> Ix ictx k
  ElimList : Ix ictx TyList
          -> (acc, x : Name)
          -> Ix ((acc, k) :: (x, TyNat) :: ictx) k
          -> Ix ictx k
          -> Ix ictx k
  ElimNat : Ix ictx TyNat
         -> (acc, x : Name)
         -> Ix ((acc, k) :: (x, TyNat) :: ictx) k
         -> Ix ictx k
         -> Ix ictx k
```

As usual the type-safety and well-scopedness of index expressions are internalized in the correct-by-construction style.

Notice that if an index expression is well-formed in some context Δ, then it should also be well-formed in the context extended with unused names e.g. in $\Delta'\Delta$. As earlier stated this is referred to as context *weakening* and was explicitly proven in listing 2.6 on page 54 for list membership proofs i.e. variables; however here weakening is over an entire expression. Intuitively this is a case of walking over the placed variables and applying weakening to the wrapped membership proofs. Unfortunately slightly more work is required due to the existence of the iterators and case expression performing implicit variable binding. In these cases weakening must be performed "under binders" which is simply a case of thinning. Recall that weakening is closely related to this slightly stronger property stating that when an index expression is well-formed in the split context $\Delta_1\Delta$, then it is well-formed in the extended context $\Delta_1\Delta_2\Delta$. To see how this helps in the case of binding, when weakening i_2 of an iterator the newly bound variables `x` and `acc` must occur at the front of the context, thus requiring all variables be placed to

its right. As an example weakening i_2 well-defined in $(x, Nat), (acc, \tau), \Delta$ by Δ' must yield a term in $(x, Nat), (acc, \tau), \Delta'\Delta$. The proof of thinning is a relatively boring tree traversal so only their signatures are given here with the full proof in appendix C.5 on page 196

```
thin : Ix (ictx1 ++ ictx) t
    -> (ictx2 : IxCtx)
    -> Ix (ictx1 ++ ictx2 ++ ictx) t
```

Proof of weakening is a direct corollary of thinning

```
weaken : Ix ictx t -> (ictx2 : IxCtx) -> Ix  (ictx2 ++ ictx) t
weaken i ictx2 = thin {ictx1=[]} i ictx2
```

Due to the high explicitness required by the EDSL to ensure terms are correctly typed by construction, a variety of thinning and weakening proofs are required throughout development. Due to their simplicity and tediousness, these proofs will generally be applied without an explicit definition. Note that because these contexts are encoded as lists of strings, or the names of the variables in scope, Idris can automatically lookup variables by name in the current context. Thus this encoding is primarily for programming convenience.

This type system above is sufficient for the introduced examples—what are the syntactic forms of $\mathscr{L}$ inhabiting these types? Answering this question is the topic of the next section.

3.2 Programs and their semantics

Having presented an initial system of types and index expressions, a type system including semantic indexing can be given for expression in $\mathscr{L}$. Note that due to the potentially large size of index expressions they are written juxtaposed with their type following notation commonly used in dependently typed programming languages, e.g. $5 : Nat\ 5$.

Note that the variables v in expressions are of a distinct *sort* from those used in index expressions since they capture expressions. In fact because all expressions are indexed by their semantics, a distinct type of context Γ is required to track variables in scope. The context must not only map variables to some type but additionally to some index expression capturing its semantics. Expression contexts are presented in figure 3.4 which follow precisely the presentation of index expression contexts, save for the addition

Contexts $\quad \Gamma ::= (x, \tau, i), \Gamma \mid \emptyset$

$\boxed{\Delta \vdash \Gamma}$ (well-formed context)

$$\frac{}{\Delta \vdash \emptyset} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash i : \tau}{\Delta \vdash (x, \tau, i), \Gamma}$$

$\boxed{\Delta; \Gamma \ni_{\mathbb{N}} (\mathbf{W}, \tau, S)}$ (well-formed context element membership)

$$\frac{}{(x, \tau, i), \Gamma \ni_0 (x, \tau)} \qquad \frac{\Delta; \Gamma \ni_w (x, \tau, i)}{\Delta; (y, \tau_2, j), \Gamma \ni_{w+1} (x, \tau, i)}$$

Figure 3.4: Contexts of expression variables.

of variables' semantic information. Note that this addition of semantic information implies an expression context Γ be well-formed with respect to an underlying index expression context Δ.

Given a notion of a well-formed variable, it's possible to present a well-formedness condition on expressions in a scoping context. An expression e is well-formed with respect to some context Γ not only in its type τ but additionally in some index expression i. Since an $\mathscr{L}$ expression's index expression captures *all* of its semantic information it's possible to define a high-powered means of automatically transporting correctness proofs from expression's index expression to a correctness proof on the expression itself. Thus expressions in $\mathscr{L}$ exist purely as a means to convey a program's *operational* intent, providing information useful in allowing for compilation to efficient machine code. The well-formedness of expressions is presented in figure 3.5.

As noted the typing rules presented in figure 3.5 exactly match those given in figure 3.3. The rules *EZero* and *ESuc* collectively state that every number has itself as a semantic index. Similarly, lists carry themselves as a semantic index. To reiterate, the empty list has the empty list index expression as its semantic index (*ENil*). Cons accepts a head with some number index expression and a tail with some list index expression and produces a list whose semantic index is the list constructed by consing this head with this tail (*ECons*).

The solely interesting rule *ECaselist* asserts that upon inspection, a list with index expression i_1 is discovered to either be empty [], or there exist some index expressions j and js which i_1 is destructed into. Thus the sub-term e_2 is checked in the extended context j, js, Δ, while the term e_3

$\boxed{\Delta; \Gamma \vdash e : \tau\ S}$ (well-formed, semantics indexed expression)

$$\text{EVar}\frac{\Delta; \Gamma \ni_v (x, \tau, i) \quad \Delta \vdash i : \tau}{\Delta; \Gamma \vdash v : \tau\ i}$$

$$\text{EZero}\frac{}{\Delta; \Gamma \vdash 0 : \mathit{Nat}\ 0} \qquad \text{ESuc}\frac{\Delta; \Gamma \vdash e : \mathit{Nat}\ i}{\Delta; \Gamma \vdash e + 1 : \mathit{Nat}\ (i + 1)}$$

$$\text{ENil}\frac{}{\Delta; \Gamma \vdash [] : \mathit{List}\ []} \qquad \text{ECons}\frac{\Delta; \Gamma \vdash e_1 : \mathit{Nat}\ i_1 \quad \Delta; \Gamma \vdash e_2 : \mathit{List}\ i_2}{\Delta; \Gamma \vdash e_1 :: e_2 : \mathit{List}\ i_1 :: i_2}$$

$$\text{ECaselist}\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathit{List}\ i_1 \\ (j, \mathit{Nat}), (\mathit{js}, \mathit{List}), \Delta; (x, \mathit{Nat}, j), (\mathit{xs}, \mathit{List}, \mathit{js}), \Gamma \vdash e_2 : \tau\ i_2 \\ \Delta; \Gamma \vdash e_3 : \tau\ i_3\end{array}}{\Delta; \Gamma \vdash \text{caselist}\ e_1\ ((x :: \mathit{xs}) \Rightarrow e_2); ([] \Rightarrow e_3) : \tau\ (\mathit{caselist}\ i_1\ ((j :: \mathit{js}) \Rightarrow i_2)([] \Rightarrow i_3))}$$

$$\text{EElimlist}\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathit{List}\ i_1 \\ (j, \mathit{Nat}), (\mathit{acc}, \tau), \Delta; (x, \mathit{Nat}, j), (\mathit{xs}, \tau, \mathit{acc}), \Gamma \vdash e_2 : \tau\ i_2 \\ \Delta; \Gamma \vdash e_3 : \tau\ i_3\end{array}}{\Delta; \Gamma \vdash \text{elimlist}\ e_1\ (x, \mathit{xs} \mapsto e_2)\ e_3 : \tau\ (\mathit{elimlist}\ i_1\ (j, \mathit{acc}) \mapsto i_2)\ i_3)}$$

$$\text{EElimnat}\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathit{Nat}\ i_1 \\ (j, \mathit{Nat}), (\mathit{acc}, \tau), \Delta; (x, \mathit{Nat}, j), (\mathit{xs}, \tau, \mathit{acc}), \Gamma \vdash e_2 : \tau\ i_2 \\ \Delta; \Gamma \vdash e_3 : \tau\ i_3\end{array}}{\Delta; \Gamma \vdash \text{elimnat}\ e_1\ (x, \mathit{xs} \mapsto e_2)\ e_3 : \tau\ (\mathit{elimnat}\ i_1\ (j, \mathit{acc}) \mapsto i_2)\ i_3)}$$

Figure 3.5: Typing rules for semantics indexed expressions.

corresponding to the empty case is checked in the usual context.

Surprisingly there's no explicit requirement on index expressions being well-formed in the expression's well-formedness condition. It is relatively easy to show that the typing relation in figure figure 3.5 guarantees that if a term is well-typed, then its semantic index is also well-formed. This fact is captured by the following soundness condition

Theorem 3.2.1 (Term-Index expression typing soundness)**.**

$$\Delta; \Gamma \vdash e : \tau\ i \implies \Delta \vdash i : \tau$$

Proof. The proof proceeds by structural induction on the shape of the typing derivation.

Case (EVar)**.** By assumption $\Delta \vdash i : \tau$.

Case (EZero)**.** By *SZero* $\Delta \vdash 0 : \mathit{Nat}$.

Case (ESuc)**.** By induction $\Delta \vdash i : \mathit{Nat}$, thus by *SSuc* $\Delta \vdash i + 1 : \mathit{Nat}$.

Case (ENil)**.** By *SNil* $\Delta \vdash [] : \mathit{Nat}$.

Case (ECons)**.** By induction $\Delta \vdash i_1 : \mathit{Nat}$ and $\Delta \vdash i_2 : \mathit{List}$, thus by *SCons* $\Delta \vdash i_1 :: i_2 : \mathit{List}$.

Case (ECaselist)**.** By induction $\Delta \vdash i_1 : \mathit{List}$, $\Delta \vdash i_3 : \tau$, and $j \mapsto \mathit{Nat}, \mathit{js} \mapsto \mathit{List}, \Delta \vdash i_2 : \tau$. Thus by *SCase* $\Delta \vdash$ caselist i_1 $(j :: \mathit{js} \Rightarrow i_2); ([] \Rightarrow i_3)$.

Case (ECasenat)**.** By induction $\Delta \vdash i_1 : \mathit{List}$, $\Delta \vdash i_3 : \tau$ holds, as does $(j, \mathit{Nat}), (\mathit{js}, \mathit{List}), \Delta \vdash i_2 : \tau$ thus by *SCaselist caselist* $i_1((j :: \mathit{js}) \Rightarrow i_2);\ ([] \Rightarrow i_3) : \tau$.

Case (EElimlist)**.** By induction $\Delta \vdash i_1 : \mathit{List}$, $\Delta \vdash i_3 : \tau$ holds, as does $(j, \mathit{Nat}), (\mathit{acc}, \tau), \Delta \vdash i_2 : \tau$ thus by *SElimlist elimlist* $i_1((x, \mathit{acc}) \mapsto i_2) i_3 : \tau$.

Case (EElimnat)**.** By induction $\Delta \vdash i_1 : \mathit{Nat}$, $\Delta \vdash i_3 : \tau$, holds, as does $(j, \mathit{Nat}), (\mathit{acc}, \tau), \Delta \vdash i_2 : \tau$ thus by *SElimnat elimlist* $i_1((x, \mathit{acc}) \mapsto i_2) i_3 : \tau$.

■

Note that in a dependently typed system no such lemma is necessary because all index expressions are guaranteed to be correct thanks to the the internalization trick from section 2.5.4 on page 47.

Although the rules from figure 3.5 are sufficient to implement the naïve, list head function, attempting to give the return value its proper index fails to typecheck.

$$\mathit{head} : \mathit{List}\ (j :: \mathit{js}) \to \mathit{Nat}\ j$$

Note that while functions aren't technically supported, the type of *head* is written with a "function type" in an abuse of notation. This is because in the implementation, functions can be defined within Idris, corresponding to macros in $\mathscr{L}$ which are inlined at every call site.

Writing down the expected program and inferring the explicit type exemplifies the problem

$$\begin{aligned} &\mathit{head} : \mathit{List}\ (j :: \mathit{js}) \to \mathit{Nat}\ (\mathit{caselist}\ (j :: \mathit{js})\ (k :: \mathit{ks} \Rightarrow k); ([] \Rightarrow 0)) \\ &\mathit{head}\ \mathit{xs} = \text{caselist}\ \mathit{xs}\ (x :: \mathit{xs}' \Rightarrow x);\ ([] \Rightarrow 0) \end{aligned} \tag{3.2}$$

There are two primary issues with the above

1. A failure to specify any reduction semantics for types precludes the possibility of reaching an intuitively expected normal form; and

2. in order to satisfy the unreachable [] case, ensuring totality, a clumsy dummy expression 0 must be introduced.

Incidentally the solutions to these problems are closely related. The strategy for solving the first problem introduces a means of type conversion by an explicit *rewrite* expression similar to the explicit coercions of System FC [WHE13]. In order to solve the second problem a means of explicit *reachability testing* is exposed allowing the programmer to interact in the process of checking similar to the typechecking scripts of Atkey [Atk15].

The above solutions require a general means of accumulating, and applying, knowledge about term index expressions. For example in order to return the correct index expression k in equation (3.2) it must be known that $j :: js = k :: ks$, otherwise x may be ascribed the incorrect index expression. Similarly, to avoid specifying a [] case, it must be shown that no substitution of $j :: js$ yields []. This knowledge will be explicitly tracked by introducing a context of facts—known constraints on term index expressions—manipulated and applied through the application of case analysis and explicit rewriting.

3.3 Index expression rewriting

The notion of index expression rewriting developed here satisfies two key properties

1. each rewriting is sound *by construction*; and

2. rewriting is performed by checking semantic, not syntactic, equivalence.

To this end a formal notion of equivalence will be constructed, $i \approx j$, along with a new syntactic form asserting that i and j are convertible given a proof of semantic equality *proof*

$$\text{rewrite } j \textit{ proof } e : \tau j \tag{3.3}$$

Note that this is an intrinsically dependently typed term. The syntactic form *rewrite* directly embeds the proof of its soundness at the application site, serving to localize the evidence which will be necessary to prove a notion of global soundness.

The notion of rewriting types by type conversion is a well-studied problem. C++ for example has an elaborate set of rules to perform automatic coercion between types e.g. at assignment sites and initialization. Object oriented languages tend to expose a few rules for coercing between primitive numeric types, e.g. performing zero or sign extension [Com99], in addition to sophisticated subtyping rules which are usually nominal [Abd13]. Such schemes are usually decidable upto parametric polymorphic functions and variance annotations [KP07].

In comparison, dependently typed languages such as Idris,[Bra13b], Agda [Nor07], Coq [Tea18], etc. tend to perform conversion by checking syntactic equality of terms' normal forms. This is due to the considerably more expressive type-system which precludes a decision procedure for testing arbitrary term equality. Consider that were this possible, the type checker could solve the halting problem. Such systems are often stratified into intensional and extensional theories [Hof95], the latter forgoing decidability of type-checking in favor of more powerful equality checking. Of particular interest in extensional theories is the inclusion of functional extensional which states that two equations agreeing everywhere are equal.

$$\forall (f\,g : A \to B)\ (x : A).f\ x = g\ x \implies f = g \quad \text{(Functional Extensionality)}$$

This allows for instance to lift proofs of propositions between functions when they are "observationally equal". While work has been done to explicitly bring this law into an intensional type theory [AMS07] we will show that with explicit rewrites the axiom can be used with impunity in the unmodified theory underlying Idris.

Convertibility via normal forms serves to reduce the problem of whether two terms are semantically equivalent, to the question of whether they are syntactically equal. In order to recapture conversion of syntactically distinct but semantically equivalent terms the usual approach is to bring the underlying judgmental equality of the system, such as that described by Martin-Löf [MS84] and used in Agda [CDP14], into the syntactic world via the following identity type

$$\frac{}{\mathit{Refl} : x = x} \tag{3.4}$$

The name *Refl* is meant to stand for "reflexive" capturing the notion that a term is only equal to itself as presented in section 2.3.1.

$$
\begin{aligned}
[\![\cdot]\!] &: Ty \to \star \\
[\![Nat]\!] &= \mathbb{N} \\
[\![List]\!] &= List\ \mathbb{N}
\end{aligned}
$$

Figure 3.6: Semantics of types

Rather than using the more automated syntactic equality induced by comparing normal forms the language $\mathscr{L}$ can be extended with a notion of semantic equality. This is made possible when a language has access to its own semantics e.g. because it is written as an embedded DSL within a suitably expressive host such as Idris. Due to the relative simplicity of such a rule, its soundness falls out as a trivial consequence as is shown later in theorem 3.5.2 on page 91.

In order to reach our goal of formalizing the rewrite rule, a notion of equivalence must be given. Thus a semantics of index expressions must first be given, allowing for the construction of a notion of index expression equivalence $i \approx j$.

3.3.1 Specifying index expression semantics

Given their relative simplicity, specifying the semantics of index expressions presented in figure 3.2 on page 70 within some variable environment is straightforward. Rather than performing substitution, a variable environment is used to track the running values, drawn from the set $\mathcal{V}$, bound to each variable. A values is either a natural number or a list of natural numbers allowing for the obvious mapping from $\mathscr{L}$ types given in figure 3.6 following the style of the STLC example.

An environment is then either empty $\emptyset$ or an extension of the environment ρ with a value $v \in \mathcal{V}$, written v, ρ. In order to ensure the correctness of variable lookup, environments must satisfy the well-formedness condition specified in figure 3.7. An environment is considered to be well-formed with respect to an underlying context Δ if for every τ typed variable $i \in \Delta$, there is a mapping from i to some v of type $[\![\tau]\!]$ in ρ. In Idris the type of environments follows precisely the formulation from the STLC, internalizing the correctness formulation.

Values $\quad \mathcal{V} ::= n \in \mathbb{N} \mid \mathcal{V} :: \mathcal{V} \mid []$

Environments $\quad \rho ::= \mathcal{V}, \rho \mid \emptyset$

$\boxed{\Delta \vdash \rho}$ (ρ is a well-formed environment)

$$\text{WfEmpty}\frac{}{\emptyset \vdash \emptyset} \qquad \text{WfCons}\frac{v \in [\![\tau]\!] \quad \Delta \vdash \rho}{(i, \tau), \Delta \vdash v, \rho}$$

Figure 3.7: Program evaluation environments

```
data IxEnv : IxCtx -> Type where
  Nil : IxEnv []
  (::) : EvalTy k -> IxEnv xs -> IxEnv ((x, k) :: xs)
```

As described earlier in section 2.5, an important notion of correctness to be shown for a traditional operational semantics is *type-safety* which was initially explored by Wright and Felleisen [WF94]. Type-safety is constituted by two theorems known as progress and preservation respectively guaranteeing that non-value terms can always reduce and that as the term is reduced, its type is invariant. As in the case of STLC leveraging dependent types and embedding the language as an EDSL allows for a simpler notion of type-safety induced by the host type system to be proven.

First by defining evaluation as a total function the system guarantees that every input index expression will result in a meaningful semantic value. Moreover since type-checking will have taken place prior to any other work, the input program will always be known to be well-typed and well-scoped. Second, by taking the evaluation function to be dependent on the type of the input, we can guarantee that the type of the input index expression is preserved in the output value. Thus type-safety becomes an intrinsic part of evaluation. Again in the style of Augustsson a tagless evaluator can be given

Listing 3.2: Semantics of index expressions

```
eval : Ix ictx t -> IxEnv ictx -> EvalTy t
eval (S i) ixEnv = S (eval i ixEnv)
eval Z ixEnv = Z
eval (Var elt) ixEnv = lookup ixEnv elt
eval [] ixEnv = []
eval (i :: is) ixEnv = eval i ixEnv :: eval is ixEnv
eval {k=k} (CaseList {k=k} i1 x xs i2 i3) ixEnv =
  listDestruct (const (EvalTy k))
               (\n, ns => eval i2 (n :: ns :: ixEnv))
```

```
                (eval i3 ixEnv)
                (eval i1 ixEnv)
eval (ElimList i1 acc x i2 i3) ixEnv =
  listFold (\n, vacc => eval i2 (vacc :: n :: ixEnv))
           (eval i3 ixEnv)
           (eval i1 ixEnv)
eval (ElimNat i1 acc x i2 i3) ixEnv =
  natFold (\n, vacc => eval i2 (vacc :: n :: ixEnv))
          (eval i3 ixEnv)
          (eval i1 ixEnv)
```

In an abuse of notation, as a notational convenience, `eval i p` may be written using the standard notation $[\![i]\!]\rho$. Evaluation proceeds in a fashion similar to the STLC evaluator with case and iterator evaluation appealing to the auxiliary functions defined in appendix C.1 on page 193. The `listDestruct` function performs dependent case analysis, however the use of a function simplifies the proof of some necessary theorems about evaluation. Empirically Idris also performs better and termination checking is aided.

Later we will show that this notion of type-safety can both be extended to semantics indexed expressions, and that it's possible to internalize this property in a way that admits the correct by construction approach to expression evaluation.

Having defined a denotation for index expressions it's possible to give a notion of semantic equivalence of index expressions with respect to an environment. Two definition are presented in figure 3.8. The first $i \approx_\rho j$ asserts that the index expressions i and j can only equated given some environment ρ mapping their variables to values. The second asserts that i is equivalent to j if they are semantically equal in every well-formed environment. Intuitively these definitions can be taken to capture the notion that the index expressions i and j are equivalent when they agree under all valid parallel substitutions of free variables tracked by the context Δ.

As suggested by the name, index expression equivalence naturally forms an equivalence relation on well-formed index expressions and environments. This follows from the fact that at its heart, index expression equivalence is an appeal to judgmental equality.

Theorem 3.3.1. *Index expression conversion $i \approx j$ forms an equivalence relation.*

The definitions of index expression equivalence within an EDSL is trivial thanks to the requirement that all index expressions are well-formed by def-

$\boxed{S \approx_\rho S}$ (index expression equivalence)

$$\text{IxEq}\frac{\begin{array}{c}\Delta \vdash \rho \\ \Delta \vdash i : \tau \quad \Delta \vdash j : \tau \\ [\![i]\!]\rho = v_i \quad [\![j]\!]\rho = v_j \\ v_i = v_j\end{array}}{\Delta \vdash i \approx_\rho j}$$

$\boxed{S \approx S}$ (universal index expression equivalence)

$$\text{IxConv}\frac{\begin{array}{c}\Delta \vdash i : \tau \quad \Delta \vdash j : \tau \\ \forall \rho. i \approx_\rho j\end{array}}{\Delta \vdash i \approx j}$$

Figure 3.8: Semantic equivalence of index expressions

inition. In Idris it's just an equality on evaluation, intuitively capturing the idea that index expression equivalence allows for rewriting in an evaluation respecting way.

```
{i, j : Ix ictx t} ->
(ixEnv : IxEnv ictx) ->
eval i ixEnv = eval j ixEnv
```

Recall that the construction of index expression equivalence was a necessary requirement towards the ultimate goal of giving a sound, general means of rewriting term index expressions.

3.3.2 A sound rewriting of index expressions

Having given the notion of term equality in figure 3.8, a means of performing correct-by-construction term rewriting can be given. Recall the language of expressions extended with a syntactic form of index expression rewriting

$$e ::= \ldots \mid \text{rewrite } S\ e \tag{3.5}$$

The typing rule for rewrite simply requires the index expressions and expression be well-formed.

$$\text{ERewrite}\frac{\begin{array}{c}i \approx j \\ \Delta \vdash j : \tau \quad \Delta; \Gamma \vdash e\ \tau\ i\end{array}}{\Delta; \Gamma \vdash \text{rewrite } j\ e : \tau\ j} \tag{3.6}$$

Of course such a rule would never be decidably satisfiable, however an EDSL can appeal to the programmer for help in proving the index expression equivalence at the application site.

The semantic equivalence for index expressions prompts a solution to the type of the function equation (3.2) on page 76, list head. However as noted type coercion is introduced to $\mathscr{L}$ with explicit syntax rather than through repeated, automatic term reduction. Given a general means of coercing between equivalent term index expressions, equation (3.2) can nearly be rewritten with the proper type

$$\begin{aligned} &\mathit{head} : \mathit{List}\ (j :: \mathit{js}) \to \mathit{Nat}\ j \\ &\mathit{head}\ \mathit{xs} = \text{rewrite}\ j\ (\text{caselist}\ \mathit{xs}\ (x :: \mathit{xs}' \Rightarrow x);\ ([] \Rightarrow 0)) \end{aligned} \tag{3.7}$$

Unfortunately attempting to give the typing derivation for this function will fail due to an inability to ascribe the index expression j to x. Consider the index expression of the case expression being rewritten

$$\text{caselist}\ (j :: \mathit{js})\ ((k :: \mathit{ks}) \Rightarrow k);\ ([] \Rightarrow 0)$$

The type of x here is the fresh variable k, not j and unfortunately nothing is known about the index expression k with which to prove this equality! The rewrite rule—in order to avoid the possibility of variables leaking outside their binder's scope—requires the source and target index expression contexts to match. Of course in this case it is obvious that a simple unification of j with k and js with ks solves the problem. However we would like to keep the core language simple, and avoid the explicit construction and application of a unification algorithm in $\mathscr{L}$.

The primary issue here is that the constraint $j :: \mathit{js} = k :: \mathit{ks}$ that is implied upon entry into the first branch of the case has been lost. Thus the problem should be solvable by explicitly tracking such knowledge. Whenever something is learned about index expressions by descending into the varying branches of cases it should be written down.

3.4 Constraint contexts

To allow for a general means of rewriting index expressions, the conversion relation $\cdot \approx_\rho \cdot$ needs access to the constraints that are implied whenever the branch of a case analysis is entered. For example given a list xs with

semantic index i, by scrutinizing xs with a case statement we learn that it is exactly empty in the [] branch, or that it can be unpacked into a head x and a tail xs'. In addition to this, something is learned about the *index expression* of xs. For instance in the [] case, xs must have zero semantic index, hence $i = []$. In the $x :: xs'$ case, xs has a semantic index $i = j :: js$ for some j and js.

At a high level, the strategy to correct the rewrite rule involves two primary steps.

1. Collect a first-order reification of the propositions constraining the shapes of index expressions. The *facts* known about index expressions.

2. Prove term equality under the *reflection* of these propositions into the rewrite rule. This serves to constrain the values in the quantified environment ρ used in evaluation.

The value of the constraint context is two-fold, not only describing extra assumptions granted to the developer during index expression equivalence checking but additionally describing the extra *obligations* evaluation must satisfy in order to maintain soundness.

3.4.1 Propositions on index expressions

The only points in the language $\mathscr{L}$ that serve to unearth information about about index expressions are the case expressions. What is learned about the shape of a term's semantic index? Consider caselist: as earlier noted the scrutinee's index expression is discovered to either be empty or some head and tail.

This motivates the simple type of constraints described in figure 3.9. The value $i \sim j$ asserts that it is safe to assume the index expressions i and j are equal. This data level representation can then be reflected into the host logic, providing either assumptions on the left of an arrow, or obligations on the right of an arrow. The constraint reflection relation is given in figure 3.10

In a dependently typed context this is a natural operation closely related to proof by reflection known as a *universe construction*. A universe construction is really just a means of coding types so they can be directly manipulated and computed over. In this case the universe of constraints trivially consists of the single "equality" code. Following the strategy of

Index expression $P ::= S \sim S$
Constraint

$\boxed{\Delta \vdash S \sim S}$ (well-formed index expression constraint)

$$\text{WfProp}\frac{\Delta \vdash i : \tau \quad \Delta \vdash j : \tau}{\Delta \vdash i \sim j}$$

Figure 3.9: Index expression constraints.

$\boxed{\Delta \vdash P \Downarrow_\rho \mathcal{V} \equiv \mathcal{V}}$ (well-formed constraint reflection)

$$\text{ReflectEq}\frac{[\![i]\!]\rho = v_1 \quad [\![j]\!]\rho = v_2 \quad v_1 = v_2}{\Delta \vdash i \sim j \Downarrow_\rho v_1 \equiv v_2}$$

Figure 3.10: Reflection of index expression constraints into equality obligations.

interpreting STLC types—also a universe construction—a function is given interpreting constraints as propositions

```
data IxConstraint : (ictx : IxCtx) -> Type where
  RequireEq : (i, j : Ix ictx t) -> IxConstraintictx

EvalConstraint : IxEnv ictx -> IxConstraint ictx -> Type
EvalConstraint ixEnv (RequireEq i j) =
  eval i ixEnv = eval j ixEnv
```

Thus the equality constraint is simply a pair of well-formed index expressions, and its reflection is the assertion that their evaluation is equal under all environments.

Equipped with a notion of index expression equivalence that can be directly manipulated at the data level, the typing rules can be augmented to track knowledge about index expressions accumulated during case analysis.

3.4.2 Type-checking with constraint contexts

Since the constraints on index expressions serve as an essential part of the rewrite rule, they consequently serve an essential role in type-checking

Constraint contexts $\quad \Xi ::= (\mathrm{S} \sim \mathrm{S}), \Xi \mid \emptyset$

$\boxed{\Delta \vdash \Xi}$ (well-formed constraint context)

$$\frac{}{\Delta \vdash \emptyset} \qquad \frac{\Delta \vdash i \sim j \quad \Delta \vdash \Xi}{\Delta \vdash (i \sim j), \Xi}$$

Figure 3.11: Constraint tracking contexts.

$\boxed{\Delta; \Xi; \Gamma \vdash e : \tau\ S}$ (well-formed expression)

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathit{List}\ i_1 \\ (j, \mathit{Nat}), (\mathit{js}, \mathit{List}), \Delta; (i_1 \sim j :: \mathit{js}), \Xi; (x, \mathit{Nat}, j), (\mathit{xs}, \mathit{List}, \mathit{js}), \Gamma \vdash e_2 : \tau\ i_2 \\ \Delta; (i_1 \sim []), \Xi; \Gamma \vdash e_3 : \tau\ i_3\end{array}}{\Delta; \Xi; \Gamma \vdash \mathrm{caselist}\ e_1\ ((x :: \mathit{xs}) \Rightarrow e_2);\ ([] \Rightarrow e_3) : \tau\ (\mathit{caselist}\ i_1\ ((j :: \mathit{js}) \Rightarrow i_2);\ ([] \Rightarrow i_3))}$$

Figure 3.12: Expressions with constraint tracking contexts.

expressions. The constraint context therefore will appear everywhere that expressions appear. The constraint context given in figure 3.11 is a simple—possibly empty—list of index expression constraints. Additionally a constraint context is considered to be well-formed with respect to the index expression context Δ when all its index expression constraints are well-formed with respect to Δ. Per usual the encoding of constraint contexts in an EDSL is trivial because there is no extra requirement on well-formedness since as ever, it only makes sense to speak of well-formed constraints. Thus a constraint context is simply a list of constraints in Δ.

```
IxConstraints : IxCtx -> Type
IxConstraints ictx = List (IxConstraint ictx)
```

Given a context for tracking index expression constraints, the typing rules for well-formed expressions can be extended to support it. In particular during case analysis the context should be extended with facts learned about the case scrutinee's index expression. The majority of the typing rules for expressions remain unchanged, figure 3.12 therefore only includes the updated case expressions; however the full set of typing rules are included in appendix C.2 on page 194. Unfortunately the addition of constraints to the EDSL highlights a pain point of internalizing the well-formedness condition. Given the natural choice of representing constraint contexts as lists, weakening must be explicitly performed any time the index expression

context is extended. While it is fortunate that the type system will point to the locations where weakening is necessary, an often significant compile time price must be paid in memory and CPU time to handle the extra evaluation burden. Such a requirement doesn't bite the relational specification because the correctness of terms in a reduced environment falls out as a natural consequence of top-level well-formedness. Thus in order to manage the known facts, expressions carry the constraint context as an additional index.

```
data Exp : IxConstraintsictx
        -> Ctxictx
        -> (t : Ty)
        -> Ix ictx t
        -> Type where
  ...
  CaseList : {ctx : Ctx ictx}
          -> Exp facts ctx TyList i1
          -> (hd, tl : Name)
          -> Exp ((RequireEq (Var Here :: Var (There Here))
                             (weaken i1 [_, _])) ::
                   weakenConstraints facts [_, _])
                  ((hd, (TyNat ** Var Here)) ::
                   (tl, (TyList ** Var (There Here))) ::
                   weaken ctx [(hd, TyNat), (tl, TyList)])
                  t'
                  i2
          -> Exp (RequireEq [] i1 :: facts) ctx t' i1
          -> Exp facts ctx t' (CaseList i1 hd tl i2 i3)
```

The additional `RequireEq` terms are really a transcription from figure 3.12. Reading the variables `Var Here` and `Var (There Here)` as De Bruijn indices indicates that the (weakened) scrutinee `i1` must have shape equal to the freshly introduced `hd` (`Var Here`) and `tl` (`Var (There Here)`). Fortunately the []case is much simpler since it is not vandalized with explicit De Bruijn tax. A possible strategy for mitigating this problem in the future is to use De Bruijn *levels* which get weakening for free trading off ease of manipulation. Another alternative is the use of various weakening constructors and only performing actual variable weakening when absolutely necessary. Such a constructor would correspond to a dependently typed equivalent of Bird and Patterson's polymorphic scoping [BP99] trick.

Equipped with a means of accessing what has been learned about the shape of a term's semantic index allows for the notion of index expression equivalence to be corrected. This in turn allows for the *head* function in

$\boxed{S \approx^{\Xi}_{\rho} S}$ (index expression equivalence)

$$\text{IxEq}\ \frac{\begin{array}{c}\Delta \vdash i : \tau \quad \Delta \vdash j : \tau \\ \Delta \vdash \rho \\ [\![i]\!]\rho = v_i \quad [\![j]\!]\rho = v_j \\ \Delta \vdash \Xi_0 \Downarrow_\rho v_0 \equiv v_{0'} \wedge \ldots \wedge \Delta \vdash \Xi_n \Downarrow_\rho v_n \equiv v_{n'} \implies v_i = v_j\end{array}}{\Delta \vdash i \approx^{\Xi}_{\rho} j}$$

$\boxed{S \approx^{\Xi} S}$ (universal index expression equivalence)

$$\text{IxConv}\ \frac{\begin{array}{c}\Delta \vdash i : \tau \quad \Delta \vdash j : \tau \\ \forall \rho . i \approx^{\Xi}_{\rho} j\end{array}}{\Delta \vdash i \approx^{\Xi} j}$$

Figure 3.13: Semantic equivalence of index expressions

equation (3.2) on page 76 to be properly reduced and to pass type-checking. Previously the obligation of primary difficulty when showing the equivalence $i \approx_\rho j$ was the semantic equality between the evaluation of i and j. The new equivalence relation given in figure 3.13 grants access to the assumption that all index expression constraints hold.

Updates to index expression equivalence materialize as the trivial addition of accepting the reflected constraints as an assumption. To simultaneously reflect all available constraints, the *all modality* `All` is used which lifts a predicate over a collection.

```
EvalConstraints : IxEnv ictx -> IxConstraints ictx -> Type
EvalConstraints ixEnv facts = All (EvalConstraint ixEnv) facts
```

`All` can be understood as the type-level equivalent of the all function in Haskell applying some predicate, and thus ensuring it holds, to every value in a list. Here `All` is applying `EvalConstraint ixEnv` to every value in the list `facts`, yielding their simultaneous reflection. Given the reflection of a constraint context, updating the rewrite rule is a simple addition of the new assumption

```
Rewrite : (j : Ix ictx t)
       -> ((ixEnv : IxEnv ictx) ->
```

```
      EvalConstraints ixEnv facts ->
      eval i ixEnv = eval j ixEnv)
-> Exp facts ctx t i
-> Exp facts ctx t j
```

Thus an i indexed expression can be converted to a j indexed expression if j and i are convertible given the accumulated constraints: `facts`.

Since the presented notion of expressions with constraints and explicit coercions is correct-by-construction, a sound notion of evaluation should naturally follow.

3.5 Expression evaluation

Having defined a means of reducing index expressions, a notion of expression evaluation can be given. Like the STLC and index expressions, expressions are evaluated in an *environment*, denoted by ζ, mapping variables to values. Since the semantic domain of expressions coincides with that of index expressions, the type of environment doesn't differ. In a traditional relational setting, expression evaluation might be presented in a big-step style with the following signature, mapping expressions to some set of values given a variable mapping ζ.

$$\Delta; \Gamma \vdash e \Downarrow_{\zeta} \mathcal{V} \tag{3.8}$$

While this relational construction of environments and evaluation doesn't carry much information the correct-by-construction EDSL is considerably more expressive, giving guarantees about the type and semantics of resulting values.

Variables must be mapped to an *indexed* semantic domain. These indexed host types capture the notion that a number is indexed by its value, and a list is indexed by its list of values. This is the same construction used in languages with less expressive type systems such as Haskell. As Haskell has enjoyed a slow expansion of its type-system programming patterns have been developed for capturing stronger invariants with e.g. the singleton pattern [MH10]; [LM13]. Such patterns serve to give a type-level reification of the runtime values effectively turning types into constraints on universally quantified kinds. As was presented in section 2.4 on page 38, the construction is closely related to synthetic dependent types. As noted earlier this relationship holds precisely because our own construction is a restricted form of synthetic dependent types. Though this is odd in a dependently typed context where

direct access to values within the type-level requires no tricks given that the EDSL uses Haskell style synthetic dependent types, these types offer a natural semantic embedding.

```
data IxNat : Nat -> Type where
  Z : IxNat Z
  S : IxNat n -> IxNat (S n)
data IxList : List Nat -> Type where
  Nil : IxList []
  (::) : IxNat x -> IxList xs -> IxList (x :: xs)
```

In order to capture the relationship of our own language's semantic indexed types and their target semantics in the host system, the environments must satisfy a well-formedness condition ensuring a mapping of variables to values which is index expression respecting. Thus as a condition a notion must be given for a type evaluated at an index

```
EvalIxTy : (t : Ty) -> EvalTy t -> Type
EvalIxTy TyNat  n  = IxNat n
EvalIxTy TyList xs = IxList xs
```

In each case a semantic reflection—n or xs—of the value targeted by evaluation is passed into the indexed type. Given these indexed types, an environment is well-formed with respect to an underlying context when each variable x in the context is mapped to a semantic index whose value is constrained by the index expression i of x

```
data Env : IxEnv ictx -> Ctx ictx -> Type where
  Nil : Env ixEnv []
  (::) : EvalIxTy t (eval i ixEnv)
      -> Env ixEnv ctx
      -> Env ixEnv ((x, (t ** i)) :: ctx)
```

This exemplifies how the `IxNat` and `IxList` types internalize an equality constraint. The environment could alternatively be structured by evaluating each type `t` and then explicating restricting the resulting value by an equality with the evaluation of `i`.

Given an environment binding an expression's free variables to index expression respecting values, it's possible to give a notion of index expression respecting expression evaluation. The criterion for soundness is an agreement between the evaluation of a term's index expression and the evaluation of expression itself.

Theorem 3.5.1 (Soundness of evaluation)**.**

$$\begin{array}{l}\Delta;\Gamma;\rho \vdash \zeta \\ \Delta;\Xi;\Gamma \vdash e : \tau \\ \Delta;\Xi;\Gamma \vdash e \Downarrow_{\zeta} v \\ [\![i]\!]\rho = v'\end{array} \implies v = v'$$

While this particular formulation of soundness will not be proven, a mechanization of its correct-by-construction transformation is given in theorem 3.5.2.

Theorem 3.5.2. *Sound evaluation*

```
evalExp : {ctx : Ctx ictx}
       -> (e : Exp facts ctx t i)
       -> (ixEnv : IxEnv ictx)
       -> (knowns : EvalConstraints ixEnv facts)
       -> (env : Env ixEnv ctx)
       -> EvalIxTy t (eval i ixEnv)
```

Proof. By matching on `e`—the expression under evaluation—numbers, lists, and variables are handled in the usual way noting that the constructors on the right hand side are all semantic values indexed by the evaluation of the input index expression `i` i.e. `eval i ixEnv`.

Case (Z)**.** `evalExp Z ixEnv known env = Z`

The right-hand value `Z` has index `Z = eval Z ixEnv` as required.

Case (S)**.** In the case of successor with index expression `S i`, the incremented expression `e` has index expression `i` thus applying `S` to its sound evaluation has index expression `S (eval i ixEnv)`.

```
evalExp (S e) ixEnv known env = S (evalExp e ixEnv known env)
```

Case (Var)**.** Although the index expression of a variable is abstract, the well-formedness of the input environment guarantees the availability of a semantic value with the correct index. Thus lookup of the variable `Elem (x, t ** i) ctx` in the environment yields a value of type `EvalIxTy t (eval i ixEnv)`.

```
evalExp (Var elt) ixEnv known env = lookup env elt
```

Case ([]). The nil list case holds trivially.

```
evalExp [] ixEnv known env = []
```

Case ((::)). Similarly to the nil case, the cons case is a direct consequence of the correctly evaluated sub-trees.

```
evalExp (e :: es) ixEnv known env =
  evalExp e ixEnv known env :: evalExp es ixEnv known env
```

Case (Rewrite). Like the case of variables, the index expression `j` resulting from an explicit coercion is abstract. Thus semantic values at `j` must be transported to values at the index `eval i ixEnv`. This justifies the definition of `Rewrite` since it ensures such a proof is available, describing how to get from the evaluated source type `eval i ixEnv` to the required target `eval j ixEnv`.

```
evalExp (Rewrite j unify e) ixEnv known env =
  rewrite sym (unify ixEnv known) in
    evalExp e ixEnv known env
```

Case (CaseList). The `CaseList` term indebts itself to sub-trees with a guarantee on the index expression of the scrutinee despite its apparent lacking of such a proof. Evaluation makes good on this promise, evaluating the scrutinee `e1` and *remembering* the result of this evaluation. This fact is passed into the correct sub-tree upon evaluation by an extension to the reflected constraint context. Thus `e1` is first evaluated, however the evaluation of its index expression is kept abstract by not matching on the result. Instead an equality proof of the evaluated index expression is explicitly introduced using the inspection pattern in listing 2.4 on page 46, however this too is kept abstract. Finally the index expression `i1` is explicitly evaluated using Idris' `with`, allowing reduction everywhere in the goal.

```
evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
  with (evalExp e1 ixEnv known env)
  evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
    | v with (inspectEval i1 ixEnv)
    evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
      | v | res with (eval i1 ixEnv)
```

See listing D.2 on page 205 for the full implementation of `inspectEval`. By matching on the result of evaluation `evalExp e1 ixEnv known env` and its corresponding index expression `eval i1 ixEnv`, the goal is refined from

a blocked case expression to the correct branch. Moreover the inspection pattern only allows one side to reduce, thus the required equality proof about the index expression is available. Suppose first that `e1` is nil so that the nil branch `e3` should be entered. The branch is either reachable or unreachable due to previously accumulated facts. If it's reachable then evaluation proceeds as usual, however in the unreachable case no term is available. Fortunately the developer has provided just the necessary contradiction to which the proof appeals, dispatching the obligation.

```
evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
  | [] | EvalIs iEmpty | [] = case e3 of
    Reachable (CanReach sub iToJ) e3' =>
      evalExp e3' ixEnv (sym iEmpty :: known) env
    Unreachable (NoReach noReach) _ _ =>
      void (noReach ixEnv known iEmpty)
```

Alternatively the result of evaluation `evalExp e1 ixEnv known env` is non-nil. This is not quite symmetric because the branches have an extended context and thus evaluation will be blocked by an application of weakening. Thus the proof of the evaluated index expression's shape is a guarantee *in the wrong context*. Of course extending and permuting a context cannot change the result of evaluation, thus by appeal to corollary 3.5.2.1 the evaluation of a thinned term in an extended environment can be reduced to the evaluation of the term in the reduced environment it references.

```
evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
  | (v :: vs) | EvalIs iCons | (w :: ws) = case e2 of
    Reachable (CanReach sub iToJ) e2' =>
      evalExp e2'
              (w :: ws :: ixEnv)
              (sym (thinKnownConsRespectsEval iCons) ::
               weakenKnowns known [w,ws])
              (v :: vs :: weaken env [w,ws])
    Unreachable (NoReach noReach) _ _ =>
      void (noReach (w :: ws :: ixEnv)
                    (weakenKnowns known [w,ws])
                    (thinKnownConsRespectsEval iCons))
```

Case (ElimList/ElimNat)**.** The final two cases evaluating list and nat iterators are proven by two further, simultaneously defined lemmas `evalElimList` lemma 3.5.2.1 and `evalElimNat` lemma 3.5.2.2.

```
evalExp (ElimList e1 x acc e2 e3) ixEnv known env =
   evalElimList {x=x}{acc=acc} ixEnv known env
```

```
                (evalExp e1 ixEnv known env)
                e2
                (evalExp e3 ixEnv known env)
evalExp (ElimNat {i1=i1} e1 x acc e2 e3) ixEnv known env =
  evalElimNat {x=x}{acc=acc} ixEnv known env
              (evalExp e1 ixEnv known env)
              e2
              (evalExp e3 ixEnv known env)
```

The complete listing of this proof is given in listing 3.3. ■

The above proof of sound evaluation appeals to two lemmas evaluating list iterators and nat iterators respectively. While the proofs themselves are relatively simple, the *propositions* are unwieldy. The strategy is to perform the usual fold operation where each step not only accumulates a new value, but additionally ensures that its new index expression agrees with the result of taking a step in the evaluated input index expression.

Lemma 3.5.2.1 (Sound list iterator evaluation)**.**

```
evalElimList : {w1 : List Nat}
             -> {w3 : EvalTy t}
             -> {facts : IxConstraints ictx}
             -> {ctx : Ctx ictx}
             -> {i2 : Ix ((acc, t) :: (x, TyNat) :: ictx) t}
             -> (ixEnv : IxEnv ictx)
             -> EvalConstraints ixEnv facts
             -> (env : Env ixEnv ctx)
             -> (v1 : EvalIxTy TyList w1)
             -> (s2 : Exp {ictx=(acc, t) :: (x, TyNat) :: ictx}
                          (weakenConstraints
                             facts
                             [(acc,t), (x,TyNat)])
                          ((acc, (t ** Var Here)) ::
                           (x, (TyNat ** (Var (There Here)))) ::
                           weaken ctx [(acc,t), (x,TyNat)])
                          t
                          i2)
             -> (v3 : EvalIxTy t w3)
             -> EvalIxTy t
                  (listElim (const (EvalTy t))
                            (\v, xs', vacc =>
                               eval i2 (vacc :: v :: ixEnv))
                            w3
                            w1)
```

Proof. The proof proceeds by evaluating the expression body and its index expression in lock-step, ensuring that evaluation of the iterator body follows the evaluation of the required index expression. Proceeding by recursion on the value to iterate over, the empty case is trivial, simply returning the input value `v3` with nil index expression.

```
evalElimList ixEnv known env [] e2 v3 = v3
```

In the non-nil case by recursion on the tail of the iterated index expression `ws` we have `iacc`, the required index expression resulting from evaluating the tail of the iterated value `vs`. Thus the obligation is to show that evaluating one more step of the body over the extended list `v :: vs` results in a value with index expression obtained from performing one step of evaluation over the extended index expression `w :: ws`.

```
evalElimList {t=t}{w1=w :: ws}{w3=w3}{i2=i2}
  ixEnv known env (v :: vs) e2 v3 =
  let iacc = listElim (const (EvalTy t))
                      (\v, xs', vacc =>
                         eval i2 (vacc :: v :: ixEnv))
                      w3 ws
  in evalExp e2
             (iacc :: w :: ixEnv)
             (weakenKnowns known [iacc ,w])
             (evalElimList ixEnv known env vs e2 v3 ::
              v ::
              weaken env [iacc, w])
```

■

Note that the proposition above is much easier read without the explicit applications of weakening. It states that given some list to iterate over `v1` with index expression `w1` and an initial accumulator value `v3` with index expression `w3`, the result of folding `v1` by the function `s2` has an index given by folding over `w1` with the function `i2`.

The proof of sound evaluation of iteration over a natural number follows precisely the shape of the proof on list iteration. The proof is given by recursion on the natural number to iterate over.

Lemma 3.5.2.2 (Sound nat iterator evaluation)**.**

```
evalElimNat : {w1 : Nat}
            -> {w3 : EvalTy t}
            -> {facts : IxConstraints ictx}
            -> {i2 : Ix ((acc, t) :: (x, TyNat) :: ictx) t}
```

```
-> {ctx : Ctx ictx}
-> (ixEnv : IxEnv ictx)
-> EvalConstraints ixEnv facts
-> (env : Env ixEnv ctx)
-> (v1 : EvalIxTy TyNat w1)
-> (e2 : Exp (weakenConstraints
                facts
                [(acc, t), (x, TyNat)])
              ((acc, (t ** Var Here)) ::
               (x, (TyNat ** Var (There Here))) ::
               weaken ctx [(acc, t), (x, TyNat)])
              t
              i2)
-> (v3 : EvalIxTy t w3)
-> EvalIxTy t
     (natElim (\value => EvalTy t)
              (\n, vacc =>
                 eval i2 (vacc :: n :: ixEnv))
              w3
              w1)
```

Proof.

```
evalElimNat ixEnv known env Z e2 v3 = v3
evalElimNat {t=t}{w1=S w}{w3=w3}{i2=i2}
  ixEnv known env (S v) e2 v3 =
  let wacc = natElim (const (EvalTy t))
                     (\v, vacc =>
                        eval i2 (vacc :: v :: ixEnv))
                     w3 w
  in evalExp e2
             (wacc :: w :: ixEnv)
             (weakenKnowns known [wacc ,w])
             (evalElimNat ixEnv known env v e2 v3 ::
              v ::
              weaken env [wacc, w])
```

■

The above proof of soundness relies on one further corollary asserting that evaluating a term is equivalent to evaluating its weakened copy in an extended environment. Intuitively this should hold because weakening simply walks over a term's variables and shifts the element membership proofs to point further into the context. Thus the proof itself is a case of walking over the variables and asserting that looking up the shifted value in

the extended environment is equal to looking up the unshifted value in the vanilla environment.

Corollary 3.5.2.1 (Weakening respects evaluation)**.**

```
weakenRespectsEval : (i : Ix ictx k)
                   -> (ixEnv1 : IxEnv ictx1)
                   -> (ixEnv : IxEnv ictx)
                   -> eval (weaken i ictx1) (ixEnv1 <++> ixEnv)
                      =
                      eval i szEn
```

Proof. The proof is a direct consequence of the stronger property that thinning respects evaluation when the left-hand side of the split context is empty.

```
weakenRespectsEval i ixEnv1 ixEnv =
  thinRespectsEval {ictx1=[]} i [] ixEnv ixEnv1
```

■

As noted although the full proof is intuitively straightforward, it's cumbersome to mechanize and is therefore relegated to lemma C.5.0.1 on page 196.

The above evaluation expression is unfortunately slow to type-check. In fact on a modern boosted 4GHz machine it has tended to take over 24 hours. Fortunately this slow function can be shortcut. Instead of evaluating the expression itself, it can be evaluated by proxy, evaluating only the expression's index expression. This is possible thanks to a powerful means of lifting proofs from evaluated index expressions to evaluated expressions using the implied equality constraint of the indexed domains. A high-level strategy for lifting proofs is presented

1. Prove that indexed semantic values can be reconstructed from their unindexed "erased" value.

2. Prove the *canonicity* property of indexed types which guarantees each set of the family is uniquely inhabited.

3. Use canonicity on two reconstructed values to transport one proof to the other.

Solving the first two problems go hand-in-hand. Fortunately reconstructing `IxNat` and `IxList` is relatively simple. However, we would like to ensure

Listing 3.3: Sound case evaluation

```
evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
  with (evalExp e1 ixEnv known env)
  evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
    | v with (inspectEval i1 ixEnv)
    evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
      | v | res with (eval i1 ixEnv)
      evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
        | [] | EvalIs iEmpty | [] = case e3 of
          Reachable (CanReach sub iToJ) e3' =>
            evalExp e3' ixEnv (sym iEmpty :: known) env
          Unreachable (NoReach noReach) _ _ =>
            void (noReach ixEnv known iEmpty)
      evalExp (CaseList {i1=i1} e1 x xs e2 e3) ixEnv known env
        | (v :: vs) | EvalIs iCons | (w :: ws) = case e2 of
          Reachable (CanReach sub iToJ) e2' =>
            evalExp e2'
                    (w :: ws :: ixEnv)
                    (sym (thinKnownConsRespectsEval iCons) ::
                     weakenKnowns known [w,ws])
                    (v :: vs :: weaken env [w,ws])
          Unreachable (NoReach noReach) _ _ =>
            void (noReach (w :: ws :: ixEnv)
                          (weakenKnowns known [w,ws])
                          (thinKnownConsRespectsEval iCons))
```

that this reconstructed value is somehow the "correct" one with respect to some input: in this case the index used for reconstruction. To understand why this strategy is suitable, it's useful to first consider the indexed values through the lens of a *bundle*.

3.5.1 Erasing and recovering semantic indices

Consider the `IxNat` type. Following Goldblatt [Gol84] this type can be visualized as a collection of sets sitting over its "index space" `Nat` in figure 3.14. In figure 3.14 for each index `n : Nat` there is a set `IxNat n` that belongs to the family of indexed nats. Moreover there is clearly a mapping *erase* from a `IxNat` to a `Nat`, namely its erasure. The family `IxNat` can then be reconstructed by considering the pre-image of *erase*; however note that in our case the pre-image of any index is trivially unique so *erase* is an

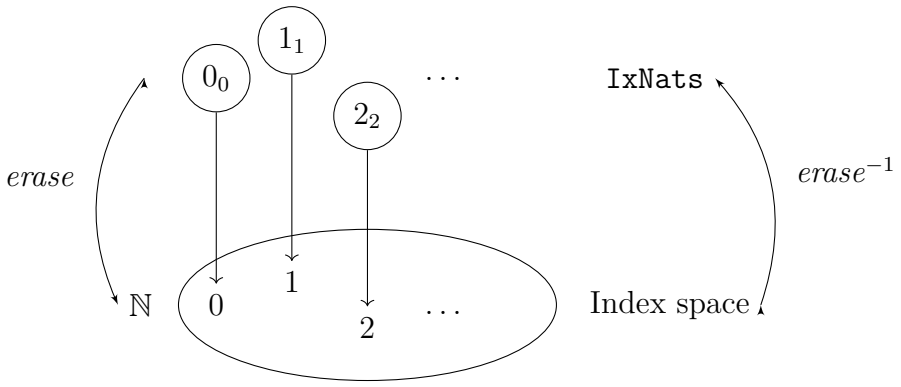


Figure 3.14: The type `IxNat`.

isomorphism! Thus we should always be able to find the exact element in the stalk space from its erasure in the base space. Taking a nat element's erasure is obvious.

```
eraseNat : IxNat i -> Nat
eraseNat {i=i} _ = i
```

Which is readily generalized to our index expression universe

```
erase : EvalIxTy t i -> EvalTy t
erase {t=TyNat}  {i=n}  _ = n
erase {t=TyList} {i=xs} _ = xs
```

Now the reconstruction function to look for is the inverse of this erasure.

```
reconstructNat : (i : Nat) -> IxNat i
reconstructNat Z = Z
reconstructNat (S n) = S (reconstructNat n)

reconstructList : (i : List Nat) -> IxList i
reconstructList [] = []
reconstructList (x :: xs) =
  reconstructNat x :: reconstructList xs

reconstructValue : (i : EvalTy t) -> EvalIxTy t i
reconstructValue {t=TyList} v = reconstructList v
reconstructValue {t=TyNat}  v = reconstructNat v
```

In order to mechanically verify this is an isomorphism, `reconstructValue` and `erase` are proven to be inverse by the following functions whose full definitions are given in appendix C.3 on page 195.

```
leftInvNat : (n : Nat) -> erase (reconstructNat n) = n
rightInvNat : (n : IxNat i)
           -> reconstructNat (erase n) = n
leftInvList : (xs : List Nat)
           -> erase (reconstructList xs) = xs
rightInvList : (js : IxList xs)
            -> reconstructList (erase js) = js
```

Thus justifying the approach of reconstructing indexed values from their indices.

The next step in efficiently evaluating expressions requires proof of the indexed value *canonicity* properties.

3.5.2 Canonical indexed values

An obvious consequence of the fact that *erase* and *reconstructValue* are bijective is that any two indexed values sharing an index are equal.

Theorem 3.5.3. *Canonicity of Indexed Values*

```
canonicalValue : (v, v' : EvalIxTy t i) -> v = v'
```

This is of course a rephrasing of the fact that *erase* is injective. The proof is a consequence of the following two lemma's proven by routine application of induction in appendix C.4 on page 195.

```
canonicalNat : (i, i' : IxNat m) -> i = i'
canonicalList : (i, i' : IxList xs) -> i = i'
```

A convenience function can also be defined proving that two values are equal when their index expressions look distinct but are known to be equal.

```
canonicalHValue : i = i'
               -> (v : EvalIxTy t i)
               -> (v' : EvalIxTy t i')
               -> v = v'
canonicalHValue Refl v v' = canonicalValue v v'
```

Note that by pattern matching on the proof of equality between `i = i'` the goal has been refined, and thus the obligation is to show `v` and `v'` are equal at *identical* semantic indices. This is sufficient for evaluating expressions.

3.5.3 Efficiently evaluating indexed expressions

The strategy to evaluate indexed expressions is to first "reconstruct" or guess the correct result value using the expression's semantic index and then to prove that this is the correct value by canonicity.

Theorem 3.5.4. *Soundness of evaluation*

```
evalExpFast : {ctx : Ctx ictx}
            -> (e : Exp facts ctx t i)
            -> (ixEnv : IxEnv ictx)
            -> (knowns : EvalConstraints ixEnv facts)
            -> (env : Env ixEnv ctx)
            -> (v : EvalIxTy t (eval i ixEnv) **
                v = evalExp e ixEnv knowns env)
evalExpFast {i=i} e ixEnv knowns env =
  let v = reconstructValue (eval i ixEnv)
  in (v ** canonicalValue v (evalExp e ixEnv knowns env))
```

Evaluating expressions in this way is really the relational specification of soundness theorem 3.5.1 on page 91 which follows from taking the erasure of both sides of the equation.

Transporting proofs from index expressions to expressions is similarly trivial. Given that evaluating two index expressions results in the same value, evaluating expressions at those indices should also produce the same value by canonicity.

```
proveEvalExpEq
  : {ctx : Ctx ictx}
 -> (e : Exp facts ctx t i)
 -> (e' : Exp facts ctx t i')
 -> (ixEnv : IxEnv ictx)
 -> (knowns : EvalConstraints ixEnv facts)
 -> (env : Env ixEnv ctx)
 -> eval i ixEnv = eval i' ixEnv
 -> evalExp e ixEnv knowns env = evalExp e' ixEnv knowns env
proveEvalExpEq e e' ixEnv knowns env evalEq =
  canonicalHValue evalEq (evalExp e ixEnv knowns env)
                         (evalExp e' ixEnv knowns env)
```

The above two proofs rigorously show that it is sufficient to talk about an expression's index expression when reasoning about its correctness. This is of primary importance for two reasons

1. first it means functional correctness can be proven in Idris at the "abstract semantics level" without the need to trip over complicated indices introduced by expressions; and

2. second, proving the soundness of transformations is reduced to proving that the transformation is index expression respecting.

The second property is particularly important because it allows for a simple means of extracting high-level properties e.g. to perform efficient compilation or to reason about performance characteristics. Consider that it might be important to compile a loop over naturals to a single addition or multiplication instruction.

Having developed a notion of evaluation, there still remains one further host feature that should be captured. Currently there is no way to explicitly mark dead code as unreachable. Given the expressive nature of the index expression language it would be convenient to reflect our static knowledge about the shape of an expression into the host system and prove that a particular branch of a case expression cannot be reached. As will be seen, this construction follows relatively naturally from the existing language architecture.

3.6 Reachability testing

Recall for a final time the body of the list head function

$$\mathit{head}\ \mathit{xs} = \text{caselist}\ \mathit{xs}\ ((x :: \mathit{xs}') \Rightarrow x);\ ([] \Rightarrow 0)$$

Given that the input is *required* to have a non-nil shape, the dummy value 0 should not need to be supplied. Moreover this should be explicitly captured at the level of the type-system with the *empty type*, the type without any values. This allows for a branch to be marked as empty, requiring the explicit construction of a contradiction to avoid the obligation of constructing a value.

We extend $\mathscr{L}$ with support for explicitly proving that the branches of a case are either *reachable* or *unreachable*. In the case of a reachable branch all the required machinery is in place, however if a branch cannot be reached then the programmer should be able to explicitly note this in the type. The syntax of cases is updated in figure 3.15 so that each branch is explicitly shown to be reachable, or unreachable and thus dead-code.

Case expressions $\quad c ::= e \mid \text{dead } S\ S$

$\boxed{\Delta; \Gamma; \Xi \vdash c : \tau\ S}$ (well-formed case expression)

$$\frac{\Delta; \Xi \vdash i \rightsquigarrow j}{\Delta; \Xi; \Gamma \vdash e : \tau\ i}$$

$$\frac{\Delta; \Xi \vdash i \not\rightsquigarrow j}{\Delta; \Xi; \Gamma \vdash \text{dead } i\ j : \tau\ k}$$

Figure 3.15: Syntax and typing rules for case expressions.

Consider in what cases it is possible for a branch to be executed. Take for example a list xs with index i scrutinized by a caselist. In the nil case we learn that i must have been [], and in the cons case we learn that i must have been some non-empty list head and tail, $j :: js$; however suppose it is known that $i = j :: js$, then clearly the nil branch should never execute! If it were to, this would imply that $[] = i = j :: js$ which is of course contradictory. Thus to show that a branch is non-executable dead-code we are obliged to prove that the scrutinee index expression and the index expression of the branch pattern are not equivalent. When two index expressions i and j are equivalent, we say that j is reachable from i and write this as $i \rightsquigarrow j$.

This of course leaves the question of what notion of equivalence should be taken. Fortunately this is precisely a *unification problem* which has been intensely studied. In order to unify two equations suitable mapping of free variables to values in the first equation must be found that yields the second equation. Such a mapping is known as a *substitution*. However rather than giving a substitution that results in the first equation being rewritten to syntactically match the second, we can request that the substitution will result in a *semantic* equality, just as with the problem of rewriting index expressions.

Defining a substitution θ is relatively straightforward, deferring to the existing notion of an index expression constraint. Substitutions are defined in figure 3.16 as constraint contexts where every constraint constrains the value of an index expression variable rather than an arbitrary index expression. Encoding the substitution restriction over a constraint context in Idris is a case of walking over the context and ensuring every constraint involves a pure variable.

```
IsSubstitution : IxConstraints ictx -> Type
```

Substitutions $\theta ::= \Xi$

$\boxed{\Delta \vdash \theta}$ (well-formed substitution)

$$\frac{}{\Delta \vdash \emptyset} \qquad \frac{\Delta \ni_w (i, \tau) \quad \Delta \vdash j : \tau}{\Delta \vdash (w \sim j), \theta}$$

$\boxed{\Delta; \Xi \vdash i \rightsquigarrow_\rho j}$ (case branch reachability)

$$\frac{\Delta \vdash \theta \quad \Delta \vdash i \approx_\rho^{\theta\Xi} j}{\Delta; \Xi \vdash i \rightsquigarrow_\rho j}$$

$$\frac{\Delta \vdash \rho \quad (\Delta \vdash \Xi_0 \Downarrow_\rho v_0 \equiv v_0' \wedge \ldots \wedge \Xi_n \Downarrow_\rho v_n \equiv v_n' \wedge [\![i]\!]\rho = [\![j]\!]\rho) \implies \bot}{\Delta; \Xi \vdash i \not\rightsquigarrow_\rho j}$$

$$\frac{\forall \rho. \Delta \vdash \rho \implies \Delta; \Xi \vdash i \not\rightsquigarrow_\rho j}{\Delta; \Xi \vdash i \not\rightsquigarrow j}$$

Figure 3.16: Case branch reachability testing

```
IsSubstitution [] = Unit
IsSubstitution (RequireEq (Var elt) j :: facts) =
  IsSubstitution facts
IsSubstitution x = Void
```

Given a substitution, the reachability of a branch is shown by proving that the scrutinee index expression and the pattern index expression are semantically equal in every index expression environment subject to the accumulated constraints Ξ extended on the left with the variable substitution θ as shown in figure 3.15. In Idris this can be captured by a type indexed by the source and destination index expression, however rather than extending the constraint context with the provided substitution, the substitution can be directly reflected as an assumption in the equality proof.

```
data IsReachable : IxConstraints ictx -> (i, j : Ix ictx t)
                -> Type where
  CanReach : (sub : IxConstraints ictx)
          -> {default () isSubst : IsSubstitution sub}
          -> ((ixEnv : IxEnv ictx) ->
              EvalConstraints ixEnv facts ->
              EvalConstraints ixEnv sub ->
              eval i ixEnv = eval j ixEnv)
```

```
          -> IsReachable facts i j
```

As an example of a reachable branch consider the cons case of the head function. In this case the scrutinee xs has non-empty index expression $j :: js$ while the pattern $x :: xs'$ has index $k :: ks$. Thus in order to show that this case is reachable, it must be shown that the equations $j :: js$ and $k :: ks$ are unifiable. Fortunately there is a relatively obvious substitution taking the index expressions of the respective head and tails to be equal: $(j \sim k), (js \sim ks), \emptyset$.

Turning to the case of a non-executable, dead-code branch we would like to know that the case scrutinee index expression, and the pattern index expression are not equivalent. In this case rather than finding a single suitable substitution, we can show that in no environment ρ are these two equations equal. In this case the substitution is effectively performed at the semantic level, where every combination of parallel substitutions is tested by the universally quantified environment. When an index expression j is not equivalent to an index expression i we say that j is unreachable from i and write this $i \not\rightsquigarrow j$. The formal definition of non-reachability is given in figure 3.16. Similar to the reachable branch, capturing this proposition and its proofs in Idris is straightforward. As usual we can simply instantiate the set of free variables by introducing an index expression environment, reflect the current constraint context, and show that were i and j equal a contradiction would follow.

```
data NotReachable : IxConstraints ictx -> (i, j : Ix ictx t)
                 -> Type where
  NoReach : ((ixEnv : IxEnv ictx) ->
             EvalConstraints ixEnv facts ->
             eval i ixEnv = eval j ixEnv ->
             Void)
         -> NotReachable facts i j
```

Thus given an explicit means of describing when a portion of code is reachable or non-reachable it's possible to encode this information directly into the EDSL syntax tree. Thus continuing in the correct-by-construction style every elided branch must not only be proven unreachable, but in the case it is not marked unreachable the developer must also show why its possible for the branch to execute.

```
data CaseBranch : (ictx : IxCtx)
               -> (facts : IxConstraints ictx)
               -> (i, j : Ix ictx t)
```

```
                -> IxConstraints ictx3
                -> Ctx ictx3
                -> (i3 : Ix ictx3 t3)
                -> Type where
Reachable : IsReachable facts i j
          -> Exp facts3 ctx3 t3 i3
          -> CaseBranch ictx facts i j facts3 ctx3 i3
Unreachable : NotReachable facts i j
            -> (ctx3 : Ctx ictx3)
            -> (i3 : Ix ictx3 t3)
            -> CaseBranch ictx facts i j facts3 ctx3 i3
```

This type states that a branch at index expression j, executed upon scrutinizing an expression with index i, is either reachable or unreachable. As usual, the contexts `ictx` and `facts` required by well-formedness are carried as indices by each case branch. Additionally the type carries a potentially extended set of constraints `facts3` accessible within the body of the branch. This corresponds to extension by any newly learned facts as was the case in the simpler `CaseList` node. Finally, in the case of an unreachable branch, the resulting context and index expression are explicitly placed to aid Idris' type inference.

Making use of this definition of a branches' reachability or non-reachability finally allows for a complete derivation of the head function's well-formedness. Consider again the head function on xs with index $i :: is$.

$$head\ xs = \text{caselist}\ xs\ ((x :: xs') \Rightarrow \text{rewrite}\ j\ x);\ ([] \Rightarrow \text{dead}\ (i :: is)\ [])$$

The well-formedness condition for dead $(i :: is)$ [] requires proof that [] can never be reached from $i :: is$ given any substitution of i and is.

$$\forall \rho.(i, Nat), (is, List), \emptyset \vdash \rho \implies$$
$$(i, Nat), (is, List), \emptyset; (i :: is \approx []), \emptyset \vdash i :: is \not\rightsquigarrow_{\rho} []$$

Although evaluating i and j in ρ yields no information, replacing a syntactic variable for a semantic one, we're equipped with a proof that $[\![i :: is]\!]\rho = [\![[]]\!]\rho$ by the reflected constraint context. By discriminating the constructors this is clearly false thus proving the goal.

3.7 Summary

In this chapter we have presented an example of how dependent types can be leveraged to implement a correct by construction notion of indexed ex-

pressions with a rich notion of index expression convertibility. The language introduced is able to leverage the type-system and pattern matching features of its host to prove the soundness of its index expression conversion, and to implement static reachability testing to eliminate dead code. In the next chapter these language features are evaluated, showing how to prove a variety of familiar properties such as the associativity of addition, and how to implement tactics in the host to automate certain semantic equalities required to rewrite some classes of term index expression.

Chapter 4

Programming and proving with $\mathscr{L}$

Embedded domain specific languages enjoy all the properties exposed by their host. In the case of languages such as Haskell this may take the form of producing highly efficient code by making use of its sophisticated rewrite system. As the language has continued to develop much stronger properties can be encoded by making use of GADT's. In the case of $\mathscr{L}$ having access to a more expressive type-system allows for very strong properties about natural numbers and lists to be captured. In particular existing libraries for proving theorems can be used. In this chapter we will show how to leverage standard generic programming patterns such as proof-by-reflection by mapping index expressions into existing constructs in Idris.

The "two-step" style of proving program correctness is similar to the "lambda-sub-singleton" language $\lambda_{\leq\{\}}$ explored by Aspinov [Asp95]. In this language typed terms $t : A$ can be given a refined type at A, written $t : \{M\}_A$, meaning that $t = M$. The language is enriched with a sub-typing relation $A \leq B$ in which a singleton $\{M\}_A$ is always considered to be a sub-type of A, i.e. $\{M\}_A \leq A$. Similar to the index expressions explored in this thesis, these singletons are used to capture the idea that a specification of the factorial function fac given by $fac : \{fac\}$ should have many different implementations. Thus this specification should be somehow distinct $fac' : \{fac\}$, instead capturing the equivalence class of functions implementing factorial. However unlike in the construction presented below which allows for semantic equality, term equality in $\lambda_{\leq\{\}}$ is defined syntactically, directly on singletons. Thus the equality $N = M$ is

short for $N : \{M\}_A$.

4.1 Warm-up: addition

As a simple warm-up to programming with $\mathscr{L}$ consider the obvious primitive recursive implementation of addition

```
addRec : Nat -> Nat -> Nat
addRec Z n = n
addRec (S m) n = S (addRec m n)
```

Since primitive recursion isn't available in $\mathscr{L}$, a fold must be used instead. Fortunately the universal property guarantees this is possible.

```
add : Nat -> Nat -> Nat
add m n = natFold (\n, acc => S acc) n m
```

We simply walk over m, adding 1 at each iteration until the end where n is placed. While the above is transcribed almost exactly into $\mathscr{L}$, recall that real arrows aren't available, instead functions are described by a scope opened in the parameters: in this case m and n.

```
plus : Ix [("m", TyNat) :: ("n", TyNat)] TyNat
plus = ElimNat (var "m") "m'" "acc" (S (var "acc")) (var "n")
```

Now it's possible to inspect and execute `plus` within Idris, consider a call with `m=2` and `n=3`.

```
*> eval plus [2,3]
5 : Nat
```

Given an index expression it's possible to implement this function at the expression level by following one's nose and reading off the index expression.

```
plusExp : Exp []
               [("i", (TyNat ** var "m"))
               ,("j", (TyNat ** var "n"))]
               TyNat
               Plus.plusIx
plusExp = ElimNat (var "i") "i'" "acc" (S (var "acc"))
                  (var "j")
```

This states `plusExp` has no knowledge about its index expressions and is defined in two arguments: `i` of abstract index expression `m`; and `j` of abstract index expression `n`. Proving the correctness of this function is a case of routine induction on `m`

```
plusIxCorrect : (m, n : Nat)
             -> eval plusIx [m, n] = m + n
plusIxCorrect Z     n = Refl
plusIxCorrect (S m) n = cong (plusIxCorrect m n)
```

Where `cong` asserts that given `eval plusIx [m, n] = m + n` by induction, it follows that `S (eval plusIx [m, n]) = S (m + n)` as required. This proof lifts to the definition of `plusExp` as well thanks to canonicity.

Although this is a sufficient notion of addition, it's not particularly convenient to program with. Instead by use of the host system's function definition a powerful macro system can be developed.

4.1.1 Programming with macros

Implementing the above addition function as a real Idris function is done in the usual way, however care must be taken in selecting the index expression variable context. Consider first the naive implementation of addition

```
plusIxM : Ix [] TyNat -> Ix [] TyNat -> Ix [] TyNat
plusIxM m n = Elimnat m "m'" "acc" (S (var "acc")) n
```

This at first seems suitable and is readily tested

```
*> eval (plusIxM (S Z) (S Z)) []
2 : Nat
```

However note that when this term is placed in a larger function its context will differ from the parent's and therefore must undergo explicit weakening. Far worse, the argument passed to `plusIxM` must undergo *strengthening*, requiring proof that its context may be shrunk which is likely not possible. Instead of declaring the minimum context required by `plusIxM`, the *most general* context should be given. In this case `plusIxM` is well-defined in any context, and therefore the correct signature is parameterized over arbitrary `ictx` with no changes to the function body.

```
plusIxM : Ix ictx TyNat -> Ix ictx TyNat
          -> Ix ictx TyNat
```

This not only avoids unnecessary and slow applications of weakening but also avoids the likely problematic need for strengthening on arguments.

Defining an expression proceeds similarly, considering the most general variable and constraint contexts.

```
plusExpM : Exp facts ctx TyNat m
```

```
          -> Exp facts ctx TyNat n
          -> Exp facts ctx TyNat (plusIxM m n)
plusExpM i j = ElimNat i "i'" "acc" (S (var "acc")) j
```

Note that this type also more intuitively states the fact that adding two values of index expressions m and n results in a value with index in their sum.

The structure of these macros is not only highly efficient but also regular enough to allow for a programming pattern of functional correctness to be developed. However before attempting to prove the correctness of `plusExpM` it's valuable to consider the case paradigm of $\mathscr{L}$.

4.2 Head and tail functions

As a warm-up to performing case analysis and reachability it's relatively easy to show that the safe head and tail functions can be implemented within our language. The strategy as described in chapter 3 on page 66 is to require a non-nil shape on the input list to scrutinize.

Recall that showing the reachability of a case branch requires producing a substitution mapping the index expression of the scrutinee to the index expression of the case branch under semantic equivalence. In our case, the index expression of the input is $i :: is$ and in the non-nil case we should show that this index can be made equal to the fresh index expression $j :: js$. Thus the concrete obligation is to provide a set of constraints on an environment in $[j, js, i, is]$ such that the evaluation of $i :: is$ is equal to the evaluation of $j :: js$. Of course the obvious solution is to identify the index expressions of the heads i, j and the index expressions of the tails is, js. This is given by the `unifyConsCons` function which applies the `RequireEq` constructor to produce the two constraints. Given these constraints, proving the equality of the evaluation is straightforward. Matching on the two equalities reflected from the supplied constraints results in refining the obligation to `eval (i :: is) ixEnv = eval (i :: is) ixEnv` which is trivially proved by the application of `Refl`.

In contrast to show that a branch of a case is unreachable, it must be proven that in no environment are the input index and the branch pattern index semantically equivalent. Thus we are provided with an environment along with the reflection of the accumulated index expression constraints `facts`, and finally a proof that the index expression of the scrutinee $i :: is$ is

equal to the index expression of the branch pattern, in this case []. Of course the nil and cons constructors of the Idris `List` type can be discriminated and contradiction is trivially provided with the `impossible` keyword.

```
safeHd : Exp facts
             {ictx=[("i", TyNat), ("is", TyList)]}
             [("xs", (TyList ** var "i" :: var "is"))]
             TyNat
             (CaseList (var "i" :: var "is")
                "j" "js" (var "j")
                Z)
safeHd {facts=facts} =
  CaseList (var "xs")
    "x" "xs'" (Reachable unifyConsCons (var "x"))
    (Unreachable notNil _ Z) where

  unifyConsCons : IsReachable
                    (weakenConstraints facts
                                       [("j", TyNat)
                                       ,("js", TyList)])
                    (var "i" :: var "is")
                    (var "j" :: var "js")
  unifyConsCons =
    CanReach
      [RequireEq (var "i") (var "j")
      ,RequireEq (var "is") (var "js")]
      (\[j,js,i,is], _, [Refl, Refl] => Refl)
  total
  notNil : NotReachable facts (var "i" :: var "is") []
  notNil = NoReach (\ixEnv, facts, Refl impossible)
```

There is a final refinement that can be made to this function which we present in listing 4.1. Note that the type of the result is a branch rather than the index expression i of the list's head. To solve this as noted an application of the semantic rewrite rule can be given. The final function `safeHd'` first rewrites the index expression of x from its fresh index expression j to the index expression of the input's head i using the following reflected constraint introduced by the case statement

```
iisEjjs : lookup ixEnv Here :: lookup ixEnv (There Here) =
          lookup ixEnv (There (There Here)) ::
          lookup ixEnv (There (There (There Here)))
```

Although it's not obvious from the type consider that the value of the environment will be `[j,js,i,is]`. Thus `Here` refers to j, `There Here` refers

to *js* etc. which is precisely the proof required by the rewrite. Explicitly, $i = j$ holds by the injectivity of the cons constructor i.e. `consInjective` which yields the pair of equalities $i = j$ and $is = js$. The final rewrite required to reduce the term to the expected type is from an explicit case index expression to the normalized index expression j. Note that the index expression of the unwritten term is

```
CaseList (var "i" :: var "is") "j" "js" (var "i") Z
```

Recall that the obligation introduced by the proposed rewrite to j is that the evaluation of the case expression in any constrained environment is equal to j. This of course holds trivially by the index expression denotation in listing 3.2 on page 80.

Listing 4.1: Safe list head

```
safeHd' : Exp facts
              {ictx=[("i", TyNat), ("is", TyList)]}
              [("xs", (TyList ** var "i" :: var "is"))]
              TyNat
              (var "i")
safeHd' {facts=facts} =
  Rewrite (var "i") (\ixEnv, cs => Refl)
    (CaseList (var "xs")
       "x" "xs'" (Reachable
                    (CanReach
                       [RequireEq (var "i") (var "j" )
                       ,RequireEq (var "is") (var "js")]
                       (\[j,js,i,is], _, [Refl,Refl] => Refl))
                    (Rewrite (var "i")
                      (\ixEnv, (iisEhdtl :: _) =>
                         fst (consInjective iisEhdtl))
                      (var "x")))
    (Unreachable (NoReach (\ixEnv, facts, Refl impossible))
                 _ Z))
```

The reachability paradigm used by cases in $\mathscr{L}$ is valuable for programming in a correct-by-construction way that allows for efficient compilation to occur. However given the expressive host language Idris, it's possible to program in the usual way, shooting first and asking questions later.

In the next section the fibonacci function will be given and only proved correct after the fact. This will additionally cover the development of a pattern for proving the correctness of macros.

4.3 Functionally correct fibonacci

While it is possible to use a term's index expression information to ensure the correctness of a function, it's also possible to write "incidentally correct" functions and provide a more traditional external proof of correctness. Perhaps unsurprisingly the function to compute the n^{th} value in the fibonacci sequence can be constructed without the need for recursion or full blown eliminators. Recall that the definition of the sequence is

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_n + F_{n+1} \end{aligned}$$

These semantics can be captured in Idris directly and can be taken to form the soundness condition

Listing 4.2: Standard recursive Fibonacci equation

```
fibRec : Nat -> Nat
fibRec Z          = Z
fibRec (S Z)      = S Z
fibRec (S (S n)) = fibRec n + fibRec (S n)
```

Implementing and proving correctness of a fibonacci function in $\mathscr{L}$ is broken down into three steps

1. Define the fibonacci sequence in the language of index expressions: `fibIx`.
2. Define fibonacci expression in $\mathscr{L}$: `fibExp`.
3. Prove that the definition of `fibIx` in the index expression language agrees everywhere with the definition in listing 4.2.

Finally the correctness of `fibExp` will follow from the canonicity property of index expression values.

4.3.1 The fibonacci index expression function

Since in $\mathscr{L}$ direct recursion is not available a fold over naturals must be used instead. This is sufficient to implement the usual linear time algorithm of the fibonacci function [Sed88]. In the linear implementation only the two

most recently calculated values are stored and used to compute a single step of the sequence i.e. $(f_n, f_{n+1}) \mapsto (f_{n+1}, f_n + f_{n+1})$. Given that $\mathscr{L}$ doesn't have tuples, a pair of elements can be packed into a list instead. Additionally because there is no pattern matching available the following two projection functions on lists are required to access f_n and f_{n+1}

```
ixFst : Ix ictx TyList -> Ix ictx TyNat
ixFst xs = CaseList xs "hd" "tl" (var "hd") Z

ixSnd : Ix ictx TyList -> Ix ictx TyNat
ixSnd xs = CaseList xs "hd1" "tl"
               (CaseList (var "tl")
                  "hd2" "tl2" (var "hd2")
                  Z)
               Z
```

Notably the above two functions are only correct for lists of at least length two. In the event the list is shorter a dummy value of zero is returned thus they must be used carefully. Given accessors for the "list packed tuple", the fibonacci function is simple

```
fibIx : Ix ictx TyNat -> Ix ictx TyList
fibIx i =
  ElimNat i "acc" "i'"
          [ixSnd (var "acc")
          ,ixFst (var "acc") `plus` ixSnd (var "acc")]
          [Z, S Z]
```

At each step the accumulator `acc` holds the two most recently computed values f_n and f_{n+1} after being initialized with the pair `[0, 1]`. As a simple sanity check Idris can be asked to check the correctness of this function on 3

```
fibTest : eval (fibIx (S (S (S Z)))) [] = [fib 3, fib 4]
fibTest = Refl
```

Using the above definition of fibonacci a proper semantic indexed expression can be given.

4.3.2 The semantic indexed fibonacci function

In dependently typed programming it is often the case that a type is so specific that the program body follows directly from its type, allowing for automated proof search tools such as the `auto` keyword in Idris and the *agsy* [LB04] tool in Agda. This is similarly the case in the setting of $\mathscr{L}$.

Defining fibonacci is an exercise in mechanically translating from the definition of `fibIx`. Satisfying the outermost layer of the index expression, an `ElimNat` on the input number should be introduced. Implementing the body is similarly trivial provided the two list projection functions. Suppose `expListFst` and `expListSnd` are given, then the definition is almost character for character equivalent

```
fibExp : Exp facts ctx TyNat i
      -> Exp facts ctx TyList (fibIx i)
fibExp n = ElimNat n "i'" "acc"
                   [expListSnd (var "acc")
                   ,expListFst (var "acc") `plus`
                    expListSnd (var "acc")]
                   [Z, S Z]
```

The definitions of `expListFst` and `expListSnd` are also simple, following almost directly the definition of the `safeHead` function given in listing 4.1 on page 113. However a notable difference is that the index expression is abstract. The function should inspect its input, returning the head if available, or a dummy value of zero otherwise.

```
expListFst : Exp facts ctx TyList (var xi)
          -> Exp facts ctx TyNat (CaseList (var xi)
                                           hd tl (var hd)
                                           Z)
```

Perhaps surprisingly `expListFst` cannot easily be given for arbitrary index expression `i`, but instead only over a variable `xi`. This is because the body will perform case analysis and therefore need to prove reachability of each case. Without knowing the shape of the `xi`, reachability cannot be shown e.g. if `i = []` then surely the reachability of the cons case cannot be proven. While perhaps a sufficiently clever implementation could attempt to normalize `i` and check its shape, this implementation opts for the simpler approach of working on a value that is known to be abstract. Referring to the index expression, it follows that the body should first perform a case on the input expression, returning the head if available or a dummy value of zero otherwise.

```
expListFst {xi=xi} {hd=hd} {tl=tl} e =
  CaseList e
    hd tl (Reachable
             (CanReach
                [RequireEq (var xi) (var hd :: var tl)]
                (\(hdv :: tlv :: ixEnv), _, [iEhdtl] =>
```

```
                  iEhdtl))
            (var hd))
      (Reachable
        (CanReach
           [RequireEq (var xi) []]
           (\ixEnv, _, [iEnil] => iEnil))
        Z)
```

Proving the reachability of the two branches is trivial. Given that the index expression of the input is abstract, the necessary shape can simply be requested as a constraint. To reach the cons branch, the index expression `xi` should be the supplied head and tail. Similarly, to reach the nil branch `xi` must itself be nil.

The function projecting the second element of a list is similar enough that only its signature is given [1].

```
expListSnd : Exp facts ctx TyList (var xi)
          -> Exp facts ctx TyNat
                (CaseList (var xi)
                   hd1 tl1 (CaseList (var tl1)
                                     hd2 tl2 (var hd2)
                                     Z)
                   Z)
```

Intuitively proving reachability of the second case on `var tl1` are trivial because like `xi`, its shape is fully abstract.

As a final step we wish to prove that this implementation is correct with respect to the inefficient implementation `fibRec` given in listing 4.2 on page 114. We will show that in order to prove the correctness of this definition of fibonacci it is sufficient to prove a simpler claim: that the definition of its index expression is correct.

4.3.3 A pattern for function soundness.

A general pattern can be given for proving the correctness of functions within the index expression language

1. The function is semi-mechanically broken down into its abstract evaluation, generalizing any recursive calls over abstract terms.

2. The abstract evaluation is shown to be correct with respect to some *specification* function in the usual way.

[1] see appendix A.2 on page 188 for the full definition.

3. The evaluation of the index expression function is formally shown to be equal to the application of its specification to the evaluation of its argument.

The first step proceeds in a similar but less disciplined fashion to proofs with recursion schemes [MFP91]; [UVP01]; [HWG13] e.g. as in the "algebra of programming" of Bird and De Moor [BM96]; [MKJ09].

As an example, in order to show that the implementation `fibIx` is correct for every input it should evaluate to an application of some specification function `fibFold`. Concretely, the type signature of the fibonacci specification theorem can be given

```
fibSpec : (i : Ix ictx TyNat)
       -> eval (fibIx i) ixEnv = fibFold (eval i ixEnv)
```

To prove the above statement following the soundness pattern from the beginning of section 4.3.3, the evaluation of `fibIx` should be given. Unfolding the initial part of evaluation by hand yields

```
  eval (fibIx i) ixEnv
= eval (ElimNat i "acc" "i'"
                  [ixSnd (var "acc")
                  ,ixFst (var "acc") `plus`
                   ixSnd (var "acc")]
                  [Z, S Z])
= natFold (\n, vacc =>
    [eval (ixSnd (var "acc")) (vacc :: n :: ixEnv)
    ,eval (ixFst (var "acc") `plus`
     ixSnd (var "acc")) (vacc :: n :: ixEnv)]
    [Z, S Z]
    (eval i ixEnv)
```

Continuing to follow the pattern, rather than unfolding the remainder of the body by brute force and yielding a large cumbersome term, instead the functions `ixFst`, `ixSnd`, and `plus` should be given a specification separately.

Specifying `ixFst`

To specify `ixFst` first perform expansion

```
eval (ixFst i) ixEnv
=
listDestruct (const Nat) (\x,xs' => x) Z
             (eval i ixEnv)
```

Following the soundness pattern the recursive call of `eval` on the abstract value `i` should be generalized. Generalizing `eval i ixEnv` to `xs` a suitable definition of evaluation is mechanically derived

```
ixFstEval : List Nat -> Nat
ixFstEval xs = listDestruct (const Nat) (\x,xs' => x) Z xs
```

Step two of the pattern requires a *specification* function be provided. While this evaluation function is relatively simple, we instead opt for a pattern matching definition perhaps more common in familiar functional languages such as Haskell

```
listFst : List Nat -> Nat
listFst (x :: _) = x
listFst _ = Z
```

Step two additionally ties the above two functions together with a proof of equality.

```
ixFstCorrect : (xs : List Nat) -> ixFstEval xs = listFst xs
ixFstCorrect (x :: _) = Refl
ixFstCorrect [] = Refl
```

By matching on `xs` Idris is able to reduce both `listDestruct` and `listFst` to values and the equality can be trivially given with `Refl`.

The third and final step of the pattern is to provide a proof specifying how the `ixFst` function unfolds under evaluation. This yields a final application of some high-level function applied to the evaluation of some abstract argument.

```
ixFstSpec : (i : Ix ictx TyList)
          -> eval (ixFst i) ixEnv = listFst (eval i ixEnv)
ixFstSpec {ixEnv=ixEnv} i = ixFstCorrect (eval i ixEnv)
```

Due to the work in step two this proof goes through trivially.

Fibonacci still relies on the second projection function and addition.

Specifying `ixSnd`

Specifying `ixSnd` is similar to `ixFst`. Mechanically unfolding evaluation and generalizing

```
ixSndEval : List Nat -> Nat
ixSndEval = listDestruct (const Nat)
                    (\x, xs' => listDestruct (const Nat)
                                  (\y, xs'' => y) Z xs')
                    Z
```

The specification is given as a pattern matching function

```
listSnd : (xs : List Nat) -> listSndEval xs = listSnd xs
listSnd (x :: y :: _) = Refl
listSnd (x :: []) = Refl
listSnd [] = Refl
```

Finally the two can be tied together by a function similar to `ixFstCorrect` given on appendix A.1 on page 188 which matches on its argument allowing Idris to evaluate.

```
ixFstSpec : (i : Ix ictx TyList)
          -> eval (ixSnd i) ixEnv = listSnd (eval i ixEnv)
ixFstSpec {ixEnv=ixEnv} i = ixSndCorrect (eval i ixEnv)
```

To expand the definition of fibonacci only the specification of plus is left to give.

Specifying `plus`

Specifying plus fails to deviate from the pattern. Unfolding its definition

```
  eval (plus i j) ixEnv
= natFold (\n, vacc => eval (S (Var Here)) (vacc :: n :: ixEnv))
          (eval i ixEnv) (eval j ixEnv)
= natFold (\n, vacc => S vacc) (eval i ixEnv) (eval j ixEnv)
```

Generalizing the two recursive calls completes the derivation

```
plusEval : (m, n : Nat) -> Nat
plusEval m n = natFold (\n, vacc => S vacc) n m
```

Proving the equality of `plusEval` with the usual addition operator is given on appendix B.1 on page 191 which is a straightforward application of induction. Finally the proof of specification follows from correctness as usual.

```
plusSpec : (i, j : Ix ictx TyNat)
         -> eval (plus i j) ixEnv = eval i ixEnv + eval j ixEnv
plusSpec {ixEnv=ixEnv} i j =
  plusCorrect (eval i ixEnv) (eval j ixEnv)
```

The above definitions allow for the fibonacci function's derivation to be completed.

4.3.4 Proving the correctness of fibonacci

Recalling the previously blocked evaluation of `fibIx`, finalizing its unfolding is relatively simple by the application the above specifications

```
    natFold (\n, vacc =>
      [eval (ixSnd (var "acc")) (vacc :: n :: ixEnv)
      ,eval (ixFst (var "acc") `plus`
       ixSnd (var "acc")) (vacc :: n :: ixEnv)]
      [Z, S Z]
      (eval i ixEnv)
  = -- { Specification of plus }
      natFold (\n, vacc =>
        [eval (ixSnd (var "acc")) (vacc :: n :: ixEnv)
        ,eval (ixFst (var "acc")) (vacc :: n :: ixEnv) +
         eval (ixSnd (var "acc")) (vacc :: n :: ixEnv)])
        [Z, S Z]
        (eval i ixEnv)
  = -- { Specification of ixSnd }
      natFold (\n, vacc =>
        [listSnd vacc
        ,eval (ixFst (var "acc")) (vacc :: n :: ixEnv) +
         listSnd vacc])
        [Z, S Z]
        (eval i ixEnv)
  = -- { Specification of ixFst }
      natFold (\n, vacc =>
        [listSnd vacc, listFst vacc + listSnd vacc])
        [Z, S Z]
        (eval i ixEnv)
```

Finally after generalizing the recursive call the evaluation function is uncovered

```
fibEval : (n : Nat) -> List Nat
fibEval n = natFold (\n, vacc =>
              [listSnd vacc, listFst vacc + listSnd vacc])
              [Z, S Z] n
```

In this case the evaluation function is arguably the canonical definition of iterative fibonacci following from the universal property of fold [Hut99]. The specification proof is therefore effectively a mechanization of the steps in the above unfolding presented in appendix A.3 on page 189 with only the signature included here

```
fibSpec : (i : Ix ictx TyNat)
       -> eval (fibIx i) ixEnv = fibFold (eval i ixEnv)
```

However the above is insufficient proof of the soundness of `fibIx`, recall that a final proof of equality with the definitive *recursive* notion of fibonacci must be given. This is a relatively trivial application of equational reasoning using Idris' `Syntax.PreOrdering` package which exposes syntactic extensions for focusing on a chain of a propositions decorated with small proofs to move between them. The idea was originally proposed by Augustsson [Aug99]. Concretely the theorem to prove is given by the following type signature

```
fibFastOk : (n : Nat) -> fibRec n = listFst (fibEval n)
```

The cases F_0 and F_1 are trivial. Intuitively the F_{n+2} case should be a simple application of induction to simplify the recursive calls F_{n+1} and F_n. Thus the final proof is simply a matter of showing that the sum of two calls to the iterative algorithm `fibEval` is equal to the next value to calculate; however this is precisely the definition of `fibEval`! This is mechanized in Idris almost verbatim

Listing 4.3: Soundness of fibonacci index expression function.

```
fibEvalSound Z = Refl
fibEvalSound (S Z) = Refl
fibEvalSound (S (S n)) =
    (fibRec (S (S n)))
  =[ Refl ]= -- { Definition of fibRec }
    (fibRec n + fibRec (S n))
  =[ rewrite fibEvalSound n in Refl ]= -- { By induction }
    (listFst (fibEval n) + fibRec (S n))
  =[ rewrite fibEvalSound (S n) in Refl ]= -- { By induction }
    (listFst (fibEval n) + listFst (fibEval (S n))
  =[ Refl ]= -- { Definition of fibEval }
    (listFst (natFold fibEvalStep [Z, S Z] n) +
     listFst (natFold fibEvalStep [Z, S Z] (S n)))
  =[ Refl ]= -- { Definition of natFold }
    (listFst (natFold fibEvalStep [Z, S Z] (S (S n))))
  =[ Refl ]= -- { Definition of fibEval }
    (listFst (fibEval (S (S n))))
  QED
```

The above proof of course is about the *index expression* of the fibonacci algorithm rather than the implementation in the expression language. Fortunately as always equipped with the canonical values theorem 3.5.3 on page 100 the correctness at the expression level holds almost immediately. Consider the type of the evaluated expression

```
evalExp (expFst (expFib i)) ixEnv known env
```

```
: EvalIxTy TyList (eval (ixFst (fibIx i)) ixEnv)
```

Using the specifications of `ixFst` and `fibIx` we arrive at the definition required by the soundness proof in listing 4.3.

```
EvalIxTy TyList (listFst (fibIx (eval i ixEnv)))
```

Thus by theorem 3.5.3 simply by looking at the type of this term it must be the case that evaluation is correct because the inhabitant of this type is unique.

This pattern of proving functional correctness was applied to an "incidentally correct" copy of fibonacci; however it readily scales to correct-by-construction functions. In the next section an encoding of products using indexed lists will be considered, along with macros for the usual projection functions and proofs of their correctness.

4.4 Products

So far, all uses of tuples have used a list based encoding, requiring discipline so as to avoid for example an attempt to project from an empty list. Such use come without a *necessary* guarantee of correctness and thus require an external proof to verify proper usage. This is due to a lack of usage of a value's semantic information captured by its index. Although there is no direct encoding of "well-typed" tuples in $\mathscr{L}$, by making use of terms' index expression information, tuples can be recovered—still by using standard lists. In Haskell tuples not only provide a performance advantage but additionally a safe means of projecting the n^{th} member which isn't possible using lists because they lack static size information at the type level. In $\mathscr{L}$ by comparison since every list can have its shape arbitrarily restricted, a pair can be typed by a list with two members having indices i and j e.g.

```
somePair : Exp facts ictx TyList [i, j]
```

Then the first element can be projected by matching on the head of the list, and the second element can be projected by matching on the head of the tail. These matches are guaranteed to succeed. The first of which succeeds for the reasons formalized in `safeHd`. The second projection succeeds because the index expression of the tail can unify with the singleton list `[j]`, the head of which clearly unifies with `j` as required. Using this reasoning, templates can be given projecting the first and second elements of list encoded pairs. The first projection is trivial

```
expFst : Exp facts ctx TyList [i, j]
      -> Exp facts ctx TyNat i
expFst {i=i}{j=j} e =
  Rewrite i
    (\ixEnv, facts => reduceCaseList {ixEnv=ixEnv})
    (CaseList e
      "x" "xs'"
      (Reachable
        (CanReach [RequireEq (var "x")   (weaken i [_,_])
                  ,RequireEq (var "xs'") [weaken j [_,_]]]
          (\(x :: xs' :: ixEnv), knowns, [xEi, xs'Ej] =>
             eqListPair (sym xEi) (sym xs'Ej)))
        (Rewrite (weaken i [_,_])
          (\ixEnv, (ijEhdtl :: _) =>
             fst (consInjective ijEhdtl))
          (var "x")))
      (Unreachable (NoReach (\ixEnv, facts, Refl impossible))
                   _ Z))
```

This largely follows the implementation of `safeHd` however because it is a template in concrete index expressions i and j, rather than a function in free variables "i" and "j" the type of reachability is considerably more interesting

```
[eval (thin i [("x", TyNat), ("xs'", TyList)])
      (x :: xs' :: ixEnv)
,eval (thin j [("x", TyNat), ("xs'", TyList)])
      (x :: xs' :: ixEnv)]
=
x :: xs'
```

Thus it must be the case that the evaluation of some arbitrary index expressions i and j are equal to the fresh index expressions x and xs' which trivially holds because these are precisely the two constraints that were requested.

Following `safeHd` a final reduction of the top-level `CaseList` index expression must be performed by `reduceCaseList`. Intuitively the resulting index expression should hold trivially because the input e has index $[i, j]$, and the first branch returns x. However note the target index expression of the rewrite on x which is a *weakened* copy of i. Thus the resulting proof of equality is not a simple assertion of reflexivity, but of weakening respecting index expression evaluation.

```
reduceCaseList : eval (CaseList [i, j]
                        "x" "xs'" (weaken i [("x", TyNat)
```

```
                                                  ,("xs'", TyList)])
                          Z) ixEnv = eval i ixEnv
reduceCastList {i=i}{ixEnv=ixEnv} =
  weakenRespectsEval i [_,_] ixEnv
```

Projecting the second element of the tuple is similar up to proofs on the reachability of case branches. Although the proofs are relatively simple they are most easily solved by goal refinement. Thus the program part of the expression is given leaving holes in positions of proofs.

```
expSnd : Exp facts ctx TyList [i, j]
      -> Exp facts ctx TyNat j
expSnd {i=i}{j=j} e =
  Rewrite j
    (\ixEnv, facts => ?crushCasesToJ)
    (CaseList e
      "x" "xs'"
      (Reachable
        (CanReach [RequireEq (var "x")   (weaken i [_,_])
                  ,RequireEq (var "xs'") [weaken j [_,_]]]
          (\(x :: xs' :: ixEnv), knowns, [xEi, xs'Ej] =>
             eqListPair (sym xEi) (sym xs'Ej)))
        (CaseList (var "xs'")
          "y" "xs''"
          (Reachable
            (CanReach [RequireEq (var "y")
                                 (weaken j [_,_,_,_])
                      ,RequireEq (var "xs''") []]
              (\(y :: xs'' :: x :: xs' :: ixEnv),
                (xxs'Eij :: knowns),
                [yEj, Refl] =>
                ?xs'ConsReachable))
            (var "y"))
          (Unreachable (NoReach (\ixEnv, facts, eq =>
            ?xs'NilUnreachable)) _ Z)))
        (Unreachable (NoReach (\ixEnv, facts, Refl impossible))
                              _ Z))
```

To solve the above holes, first the reachability clauses are proven followed by the top-level refinement. The reachability in the first hole intuitively follows because the tail `[j]` of `[i, j]` is of course non-nil.

Solving `?xs'ConsReachable`

The type of the hole `?xs'ConsReachable` asserts that evaluating the tail variable produces a non-empty list `lookup ixEnv (There Here) = [y]`. Recall that under each branch two fresh variables for the respective index expressions of the list head and tail are introduced. This allows for the environment to be unpacked into at least four values: unpacking the environment to `y :: xs'' ...` further refines the goal to `xs' = [y]`. This goal is solved relatively easily by matching on the reflected constraints. In fact by matching on the constraints `var "xs''" = []` Idris has silently refined the right hand side of the equality in the goal to `[y]` rather than the slightly more cumbersome `y :: xs''`. In order to satisfy the obligation it should be asked what is known about `xs'` and `y`? Since `y` is fresh, only the reflected constraint `y = j` is known. Additionally although no constraints were explicitly introduced on `xs'`, from the initial successful match on `e` the constraint `x :: xs' = [i, j]` was pushed onto the context which is precisely what is required! Thus unpacking the head of the constraint context `xxs'Eij` brings this evidence into the proof context. The final proof is thus a simple exercise in rewriting. First the tail equality from `xxs'Eij` is projected yielding proof the proposition `xs' = [j]`, thus the goal follows from the reflected constraint `y = j`. Concretely the goal is satisfied by

```
?xs'ConsReachable = rewrite yEj in
                      snd (consInjective xxs'Eij)
```

The next hole in `expSnd` captures fact that nil cannot be reached from the tail of `[i, j]`. Of course this follows from the fact that the cons of `j :: []` is discriminated from `[]`.

Solving `?xs'NilUnreachable`

Proving that the nil case is unreachable from casing on the tail of `"xs'"` follows from the fact that `xs' = tail [i, j]` which is clearly `[j]`. Concretely the type of the hole `?xs'NilUnreachable` is cumbersome.

```
NotReachable
  (RequireEq (var "y" :: var "xs''")
             [weaken i [("x", TyNat), ("xs'", TyList)]
             ,weaken j [("x", TyNat), ("xs'", TyList)]] ::
             weakenConstraints facts [("x", TyNat)
                                     ,("xs'", TyList)])
  (var "xs'")
```

```
[]
```

Which is more easily understood as

$$y :: xs' = i :: j \wedge xs' = [] \implies \text{False}$$

Of course by the injectivity of the list constructor this implies that $xs' = [j]$ which by the additional assumption about xs' gives the obviously contradictory statement $[] = [j]$. This reasoning is directly given in Idris by an explicit application of transitivity

```
?xs'NilUnreachable = consNotNil (sym xs'Enil `trans`
                                 snd (consInjective yxs'Eij)))
```

Where `consNotNil` simply discriminates between nil and cons using an `impossible` pattern.

Having proven both reachability holes the only missing piece left is to perform the rewrite.

Solving `?crushCasesToJ`

The requirement placed on `?crushCasesToJ` involves proving that the two case statements yielding the head of the tail reduce in the obvious way, i.e.

```
eval (CaseList [i, j]
        "x" "xs'" (CaseList (var "xs'")
                            "y" "xs''" (var "y")
                            Z)
        Z) ixEnv = eval j ixEnv
```

Fortunately since no opaque definitions have been used Idris will prove this fact for us by normalization, the proof is a trivial application of reflexivity.

```
?crushCasesToJ = Refl
```

So far the use of the host system Idris has proven an effective means of performing theorem proving to ensure functional correctness of macro-defined functions. However thus far all definitions have been first-order. Incidentally by appealing again to the host Idris it's possible to define a sort of higher-order function in $\mathscr{L}$ and prove their correctness which is the topic of the next section.

4.5 Higher-order functions

Thus far, $\mathscr{L}$ programs have been written in a purely first order fashion. It is immediately apparent that the host language function support can be used to encode a sort of macro. Less apparent is the ability to use the host's support for higher-order functions similar to C's non-capturing function pointers e.g. used by FreeBSD [Mai18] and Linux [CRK05]. Although variable capture is not strictly supported, it can be simulated by defining a "higher-order macro" parametric in some larger-than-necessary index expression context ensuring the necessary variables never go out of scope. In this section, an example of the common `map` function is implemented. The function `map` applies some given function to every element of a list. In Idris the function of interest is

```
listMap : (a -> b) -> List a -> List b
listMap f [] = []
listMap f (x :: xs) = f x :: listMap f xs
```

As is now standard practice to encode the above, a map function at the index expression level must first be given.

```
mapIx : (x : Name)
     -> (Ix ((x, TyNat) :: ictx) TyNat
     -> Ix ((x, TyNat) :: ictx) TyNat)
     -> Ix ictx TyList
     -> Ix ictx TyList
mapIx x f i = ElimList i "mapIxAcc" x
                (weaken (f (var x)) [_] ::
                 var "mapIxAcc") []
```

The above encoding of map is the relatively standard encoding of `map` as a fold, guaranteed by the universal property of fold. At each step the transformed list tail is stored in `acc`. Then in order to transform the new head `x` the function `f` is applied. Finally in order to transform the empty list, no work is required so `[]` is passed as the initial value of `acc`. The abstract evaluation of this function is given by

```
   eval (mapIx x acc f i) ixEnv
= eval (ElimNat i acc x
                  (f (var x) :: var acc)
                  [])
= listFold (\n, xs =>
    (eval (f (var x)) (n :: ixEnv) :: xs)
    []
```

```
    (eval i ixEnv)
```

By routine generalization the term `eval i ixEnv` should be abstracted leaving the question of how to treat the recursive call on the higher-order function. Intuitively we would like to evaluate under the application yielding the term

```
eval (f (reifyNatIx (eval (var x) (n :: ixEnv))))
     (n :: ixEnv)
```

which reduces to

```
eval (f (reifyNatIx n)) (n :: ixEnv)
```

given some embedding function

```
reifyNatIx : (n : Nat) -> Ix ictx TyNat
```

Unfortunately this intuition would be misplaced because `f` is not actually a function, but a macro operating on syntax! The body of `f` depends on the *shape* of the input which it can unpack and inspect allowing for example performing a different operation depending on whether a genuine variable or a literal has been passed as an argument. This is at the heart of the problem solved by PHOAS [WW03] which ensures that arguments are treated parametrically. Fortunately this doesn't ultimately cause a problem because the function definition will always be available at the top-level allowing for full-blown evaluation to occur. Thus the approach taken is to abstract over a function in the types of the extended environment at the point of function application. For example, in the evaluation of `mapIx` above the function application occurs in some environment `ixEnv` extended with some list element `x`. This gives a definition of the evaluation function that looks very close to regular map which will then be applied to the evaluation of `f` applied to the unevaluated argument.

```
mapIxEval : (Nat -> Nat) -> List Nat -> List Nat
mapIxEval f xs = listFold (\n, xs' => f n :: xs') [] xs
```

By further mindless appeal to the pattern a correctness proof is given by induction with respect to the usual notion of map

```
mapIxCorrect : (f : Nat -> Nat) -> (xs : List Nat)
            -> mapIxEval f xs = map f xs
mapIxCorrect f [] = Refl
mapIxCorrect f (x :: xs) = cong (mapIxCorrect f xs)
```

The induction hypothesis `mapIxCorrect f xs` asserts that mapping over the tail is correct `mapIxEval f xs = map f xs` and thus by definition this holds after cons'ing the head `x` given by a call to `cong`. Given the correctness proof, the specification of `mapIx` is broken into two parts: first the evaluation is shown to be equal to `mapIxEval` without appealing to weakness[2], second is the usual correctness using `mapIxCorrect`.

```
mapIxEvalEq
  : (f : Ix ((x, TyNat) :: ictx) TyNat
 -> Ix ((x, TyNat) :: ictx) TyNat)
 -> (i : Ix ictx TyList)
 -> eval (mapIx x f i) ixEnv
    =
    mapIxEval (\n => eval (f (Var Here))(n :: ixEnv))
              (eval i ixEnv)
mapIxEvalEq {x=x} {ixEnv=ixEnv} f i =
  cong {f=\v => v :: zs}
    (listFoldFSubst
      {f=\y, ys => eval (weaken (f (Var Here)) [_] ::
                         var "mapIxAcc")
                         (ys :: y :: ixEnv)}
      {g=\y, ys => eval (f (Var Here)) (y :: ixEnv) :: ys}
      (\y, ys => weakenRespectsEval (f (Var Here)) [ys]
                                    (y :: ixEnv))
      (eval i ixEnv)
      [])
```

This proof is a complicated means of showing that evaluating a function body weakened under a loop is equal to evaluating that body in a reduced environment. Specifically because f doesn't need access to the accumulated list ys it's safe to evaluate it without ys in the environment. The proof that performs the actual lifting of the body equality to the loop is `listFoldFSubst` given in listing D.4 on page 206. This states that folds using stepping functions that agree everywhere are equal which intuitively is true by considering repeatedly unfolding the fold.

Using this lemma proving functional correctness is a simple case of transitivity since evaluation yields `mapIxEval` which is known to be correct with respect to `map`.

```
mapIxSpec : (f : Ix ((x, TyNat) :: ictx) TyNat
         -> Ix ((x, TyNat) :: ictx) TyNat)
```

[2]Surprisingly this proof needs to be further broken down to help direct elaboration however the result is equivalent to a decorated copy of the simpler listing above.

```
          -> (i : Ix ictx TyList)
          -> eval (mapIx x f i) ixEnv
             =
             map (\n => eval (f (Var Here)) (n :: ixEnv))
                 (eval i ixEnv)
mapIxSpec {ixEnv=ixEnv} {x=x} f i =
  mapIxEvalEq f i `trans`
  mapIxCorrect (\n => eval (f (Var Here)) (n :: ixEnv))
               (eval i ixEnv)
```

Exactly as expected, the proof of correctness can be given up to the application of the macro to its variable.

Given the correctness of the index expression function it's possible to give a verified implementation of a higher-order function in the expression language in the usual style.

4.6 Automated index expression rewriting

Recall that proof by reflection is a strategy for proving a broad spectrum of properties about a structure by mapping it into a first order representation that can be directly inspected and manipulated. For example highly regular structures such as ring, groups, and monoids admit a *normalization procedure* which allows for a decision procedure to automatically test if two equations are equal. Automated ring solvers have long been used in e.g. Coq [GM05], Idris [SB17], and Agda. Using the generic, constructively verified monoid solver from section 2.7.1 on page 56 which tests the equality of two terms by right associating all parentheses some rewrites on index expressions can be automated.

In this section a tactic for automatically rewriting expressions whose index expressions are sums. This corresponds to performing rewrites under the monoid formed by addition with the zero unit. Thus the concrete goal is that given two expressions with indices `i` and `j`, give a proof that evaluation agrees `eval i ixEnv = eval j ixEnv`. The development of this tactic proceeds in two stages

1. First an explicit addition monoid is given.

2. A copy of the solver function is instantiated with the correct monoid.

3. A prover function is provided, *presuming* that the solver succeeded, allowing for the underlying proof of equality to be projected while maintaining totality.

Finally an example is given showing how to use the rewriting function.

4.6.1 A monoid of addition

Recall that a monoid is some set A equipped with an associative binary operation $\cdot \times \cdot$, and a unit value ϵ satisfying the laws $\forall a \in A. \epsilon \times a = a \times \epsilon = a$. Clearly addition on the natural numbers with unit zero forms a monoid; however rather than using the standard definition of addition exposed by Idris, later work is avoided by taking the evaluation of a macro `plus` in the index expression language.

```
plus : Ix ictx TyNat -> Ix ictx TyNat -> Ix ictx TyNat
plus i j = ElimNat i "acc" "i'" (S (var "acc")) j

plusEval : Nat -> Nat -> Nat
plusEval m n = natFold (\n,acc => S acc) n m
```

As usual this definition's correctness is guaranteed by the universal property of fold.

```
plusEvalCorrect : (m, n : Nat) -> plusEval m n = m + n
plusEvalCorrect Z     n = Refl
plusEvalCorrect (S m) n = cong (plusEvalCorrect m n)
```

Given this notion of addition, proof it forms a monoid is proof of associativity and the unit laws

```
plusEvalZeroRightNeutral : (m : Nat) -> m `plusEval` Z = m
plusEvalAssociative : (m, n, k : Nat)
                   -> (m `plusEval` n) `plusEval` k =
                      m `plusEval` (n `plusEval` k)
```

Intuitively these laws hold by repeated invocation of the corresponding law on built-in addition, followed by substitutions with `plusEvalCorrect`[3]. The full mechanization is given in listing B.1 on page 191.

[3]Functional extensionality could also be used to provide an automatic lifting from proofs on `(+)` to proofs on `plusEval`. While this is guaranteed to cause no issue with explicit coercions, it does mean that Idris will no longer be able to reduce, and automatically prove simple properties.

The final unit law $\forall a.0 + a = a$ holds by definition because addition is specified recursively on its first argument.

In the interest of generality, the monoid solver is defined in terms of an abstract monoid interface: `ExplicitMonoid` given in appendix B.3 on page 192. Thus the plus monoid follows

```
implementation [PlusExplicitMonoid] ExplicitMonoid Nat where
  unit = Z
  append = plusEval
  appendAssociative i j k = plusEvalAssociative i j k
  unitLeft i = Refl
  unitRight i = plusEvalZeroRightNeutral i
```

Equipped with an explicit monoid we move to the next step in automatically rewriting equations: instantiating the solver function with the above monoid.

```
eqPlusEvalExp : MonoidExp PlusExplicitMonoid ctx m
             -> MonoidExp PlusExplicitMonoid ctx n
             -> Maybe (m = n)
eqPlusEvalExp e1 e2 = eqMonoidExp e1 e2
```

The final step is to gain access to the underlying equality, projecting out of the `Maybe` monad. This is necessary because the rewrite rule expects an equality. Intuitively this is no problem because the shape of the inputs are statically known, and thus the solver will only be called on inputs for which we know it will succeed. As usual this is done with a type restricting the shape of the result. Idris provides the following type asserting that a value in `Maybe` is non-empty

```
data IsJust : Maybe a -> Type where
  ItIsJust : IsJust (Just x)
```

To project the equality, simply match on a proof of this type and the non-empty branch is inaccessible.

```
provePlusEvalExp : (ctx : List Nat)
                -> (e1 : MonoidExp PlusExplicitMonoid ctx m)
                -> (e2 : MonoidExp PlusExplicitMonoid ctx n)
                -> {default ItIsJust
                    mEn : IsJust (eqPlusEvalExp e1 e2)}
                -> m = n
provePlusEvalExp ctx e1 e2 with (eqPlusEvalExp e1 e2)
  provePlusEvalExp ctx e1 e2 {mEn=ItIsJust} | Just eq = eq
```

The use of `default` requests Idris apply this constructor to solve the hole. Note that the computation `eqPlusEvalExp e1 e2` is performed despite the fact that it could have been extracted from the proof of its success. This is because `ItIsJust` should carry no computational value, thus allowing its erasure at compile time.

The final step is of course application of `provePlusEvalExp` in a useful way. Consider proof of $(i_1 + i_2) + (i_3 + i_4) = i_1 + (i_2 + i_3) + i_4$. This first requires left reassociation, followed by right reassociation to get $(i_2 + i_3)$; Moreover gaining access to the right term is often insufficiently automatic with `rewrite` requiring an explicit application of substitution, repeatedly rewriting the term as in proof of `plusEvalFoldAssoc` in listing B.1 on page 191. Instead `provePlusEvalExp` can now be used to provide a rewrite function.

```
sum : Exp [] [] TyNat
          ((var "i1" `plus` var "i2") `plus`
           (var "i3" `plus` var "i4"))
   -> Exp [] [] TyNat
          ((var "i1" `plus` (var "i2" `plus` var "i3"))
           `plus` var "i4")
```

Although the body is a single application of the `Rewrite` rule, the reflection of the terms to prove equal must still be explicitly written using the monoid expression constructors.

```
sum e =
  Rewrite ((var "i1" `plus` (var "i2" `plus` var "i3")) `plus`
           var "i4")
          (\[i1,i2,i3,i4], [] =>
             provePlusFoldExp [i1,i2,i3,i4]
               ((Var Here `App` Var (There Here)) `App`
                (Var (There (There Here)) `App`
                 Var (There (There (There Here)))))
               ((Var Here `App`
                (Var (There Here) `App`
                 Var (There (There Here)))) `App`
                Var (There (There (There Here)))))
          e
```

The automated prover is not restricted to terms in variables. Any arbitrarily complicated argument to `plus` can be encoded by a variable and placed in the context. In fact, the context here `[i1,i2,i3,i4]` is not made up of variables, but their evaluation in the index expression environment.

4.7 Summary

This chapter presented several examples of how to program in $\mathscr{L}$. Additionally programming paradigms were developed, showing how the development of programs proceed. We motivate a style where first a program's semantics are defined as a distinct Idris function, followed by an implementation written by "reading off" the type up to unreachable cases. This chapter additionally developed a programming *pattern* for proving the correctness of functions given in $\mathscr{L}$ semi-directly in Idris via their index expression. The pattern describes a mechanical means of unrolling a definition and then reasoning directly about its *evaluation function*. This correctness proof is additionally known to hold for the program implementation by a simple application of the canonical values theorem.

In the next chapter a static analysis is explored, showing how the host's type system can be used to simplify the specification and correctness of the analysis. Finally the results of the static analysis are used to compile $\mathscr{L}$ programs directly into C. This compilation will make direct use of the case reachability information during code emission as a verified optimization pass. As usual the compilation procedure will use the correct-by-construction strategy to ensure it is type and scope-respecting.

Chapter 5

Verified Cost Analysis

In this chapter the index expression of a term is used to direct a correct by construction program transformation. The transformation performs a structural lifting on terms to optimize the use of numbers and literals similar to the `ExTT` optimization in Epigram [Bra05b]. Thus presenting a formally verified means of decorating an "operationally friendly representation" with its intended semantics in the correct-by-construction style. Such decorations are available to the programmer in the Agda language where pragma's are available to bind a function to a more efficient internal representation. The transformed tree is then used to exemplify the use of index expressions in directing static reasoning of program execution costs. Finally it is shown precisely how the transformed representation supports the emission of efficient low-level code by constructing, and proving the correctness of, a compilation down to C, via a low-level virtual machine.

The costing of an input will operate over a mixture of *structural* and *non-structural* elements requiring an expression be decomposed prior to analysis. Thus the development of the cost analysis on expression will be broken down into two primary stages

1. Develop a cost analysis over a language of statements by leveraging the host's notion of normalization.

2. Give a decomposition of the expression language $\mathscr{L}$ from chapter 3 to the costable statement language using reflection.

Therefore to begin with, a language of statements $\mathscr{S}$ is developed along with its corresponding cost analysis.

5.1 A structured statement language

Building upon the work in chapter chapter 3 index expressions can be used to aid in both verifying the correctness of structural program transformations and in the static analyses of terms. Incidentally these two operations go hand-in-hand because analysis can be performed over some type of transformed graph. In this case, to cost programs we would like not to reason about their semantics, but about the structural aspects that will be efficiently mapped to machine code. For example if efficiency is in mind, it is a poor idea to compile every addition operation to a loop when a single instruction suffices. Instead constituent expressions in $\mathscr{L}$ are considered to be in one of two categories:

1. structurally executed *statements*; and
2. atomically executed *expressions*.

In reality the atomically executed expressions will likely not be atomic at the ISA level, however from a cost perspective they will be analyzed as a single unit.

Thus statements can be thought of as structured *control flow* operations that thread together expressions by performing e.g. sequencing, looping, branching etc. Such operations will have a cost that depends on their sequences' respective index expressions such as the number of times a loop body is executed, or the cost of a particular branch that is taken in a case statement. Note that having static semantic information is a crucial element required to give reasonably exact costs to control flow operations. Moreover because costs are taken over *evaluated* index expressions, i.e. values in the host, properties can be proven over bounded inputs using the usual theorem proving machinery exposed by the host.

In contrast, atomic expressions are special *intrinsic* operations that are given a cost by some external oracle that does not rely on the precise expression itself. This allows for the capturing of the idea that the underlying compiler or processor is able to map large expressions into a variety of different instruction sequences or to optimize its evaluation with out-of-order execution in a way that is opaque to the analysis.

To begin with the partitioned language of structured control flow statements and their atomically costed expressions is presented in figure 5.1.

Atomic expressions	$a ::= a + a \mid n \in \mathbb{N} \mid v \in \mathbb{N}$
Case branches	$b ::= s \mid \text{dead } S\ S$
Statements	$s ::= a$
	$\mid \text{rewrite } j\ p\ s$
	$\mid []$
	$\mid s :: s$
	$\mid \text{let } x = s \text{ in } s$
	$\mid \text{caselist } s\ ((x :: xs) \Rightarrow b); ([] \Rightarrow b)$
	$\mid \text{elimlist } s\ (x, acc \mapsto s)\ s$
	$\mid \text{elimnat } s\ (x, acc \mapsto s)\ s$

Figure 5.1: Language of statements and expressions

Expressions consist of addition of natural numbers and placement of variables. Note that expressions don't include lists or actual operations on lists such as e.g. inner product. Expressions are meant to be somehow sufficiently "low-level" to be atomically costed. Given that well-formed lists will be indexed, it would in fact be possible to include a subset of operations on 3-tuples or 4-tuples which might be viewed as packed $xyzw$ vectors [CW00]. Such operations could then capture the idea that packed numbers can be efficiently mapped down into SIMD intrinsics/instructions such as SSE and AVX on amd64 [Cor16] architectures, or NEON [Hol14] on armv7 and armv8 architectures.

The statements presented in figure 5.1 closely mirror the expression language developed in chapter 3, having number and list cases, in addition to loops over lists and numbers. Notably, local variable binding via let is added. This is of course a requirement if the results of statements are to take part in expressions—without let consider the impossible task of summing two loops.

Suppose we are given a heuristic $h^\star$ for predicting the cost of an atomic expression a. How can this heuristic be structurally extended to cost programs in the full statement language? Only two of the cases necessary for consideration are difficult: how to cost a case analysis, and how to cost a loop. In the WCET community there exist sophisticated, automated strate-

gies for predicting loop bounds [Xi07]; [Wil+08] e.g. by symbolic execution, by the generation of prototypical input data [Wil+08], and by the construction and solving of recurrence relations such as with COSTA [Alb+07] or PURRS [Alb+07]. Similar strategies can be employed to determine the worst case cost of a conditional statement.

In contrast given that the semantics of variables are available, an exact cost analysis can be given in the style of Brady and Hammond [BH05], and Crary and Weirich [CW00]. As will be shown this is akin to symbolic execution using the host's normalization procedure. For example because the index expression of a loop bound is always available, calculating the exact cost as a function of the cost of the loop body is trivial. Similarly rather than taking a coarse upper bound on the cost of a conditional—e.g. by considering the cost to be the worst of the two branches—the cost itself can be conditional, matching the shape of the underlying statement. In this way the cost of a branching statement can only be reduced once the index expression of the expression under scrutiny by the case is known. Thus the cost of an expression is induced by its *trace* in a given index expression environments. Given an atomic expression costing heuristic $h^\star$ a syntax-directed directed cost relation $\Delta; \Gamma \vdash s \Uparrow_\rho C$ is presented in figure 5.2 capturing the total cost C as the sum of the individual steps required to evaluate a program s.

Many of the rules such as MkConsCost and LetCost intuitively take the cost of their two components and sum them. Performing a type coercion with a rewrite of course requires no runtime cost and thus the RewriteCost simply returns the cost of the underlying statement being coerced. Costing atomic expressions is similarly simple, given directly by the function $h^\star$ as defined by the AtomicCost rule.

The cost of case expressions and iteration are more interesting. Handling caselist is broken into three cases corresponding to the branch reachability information which is immediately available from the semantic index. To understand why three cases are necessary, first consider the cost of evaluating a case over a list known to be nil. Such a list will only ever evaluate the nil branch. However if a list is known to be non-nil, the scrutinee *must still be evaluated*. This is because the head and tail need to be made available for use in the branch and thus casing over e.g. a complex expression resulting in a list must be reduced. In the final case, when the list is not known to be nil or non-nil, first the scrutinee must be evaluated, and then *either* the nil or non-nil branch must be evaluated. Consider each case in details:

$\boxed{\Delta; \Gamma \vdash s \Uparrow_\rho C}$ (traced statement cost)

$$\text{AtomicCost}\ \frac{C = h^\star(a)}{\Delta; \Xi; \Gamma \vdash a \Uparrow_\rho C}$$

$$\text{RewriteCost}\ \frac{\Delta; \Xi; \Gamma \vdash s \Uparrow_\rho C}{\Delta; \Xi; \Gamma \vdash \text{rewrite}\ j\ p\ s \Uparrow_\rho C}$$

$$\text{MkNilCost}\ \frac{}{\Delta; \Xi; \Gamma \vdash [] \Uparrow_\rho 0} \qquad \text{MkConsCost}\ \frac{\Delta; \Xi; \Gamma \vdash s_1 \Uparrow_\rho C_1 \quad \Delta; \Xi; \Gamma \vdash s_2 \Uparrow_\rho C_2}{\Delta; \Xi; \Gamma \vdash s_1 :: s_2 \Uparrow_\rho C_1 + C_2}$$

$$\text{LetCost}\ \frac{\Delta; \Xi; (x, \tau_2, i_2), \Gamma \vdash s_2 \Uparrow_\rho C_2 \quad \Delta; \Xi; \Gamma \vdash s_1 \Uparrow_\rho C_1}{\Delta; \Xi; \Gamma \vdash \text{let}\ x = s_1\ \text{in}\ s_2 \Uparrow_\rho C_1 + C_2}$$

$$\text{ElimNatCost}\ \frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash s_1 \Uparrow_\rho C_1 \quad \Delta; \Xi; \Gamma \vdash s_3 \Uparrow_\rho C_3 \\ \Delta; \Xi; \Gamma \vdash s_1 : \mathit{Nat}\ i_1 \quad \Delta; \Xi; \Gamma \vdash s_3 : \tau\ i_3 \\ [\![i_1]\!]\rho = v_1 \quad [\![i_3]\!]\rho = v_3 \\ [\![i_2]\!](v_3, v_1, \rho) = v' \\ \Delta; \Xi; \Gamma \vdash s_2 \Uparrow_\rho^{v_1, v_3} v', C'\end{array}}{\Delta; \Xi; \Gamma \vdash \text{elimnat}\ s_1\ (x, \mathit{acc} \mapsto s_2)\ s_3 \Uparrow_\rho C_1 + C_3 + C'}$$

$$\text{ElimListCost}\ \frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash s_1 \Uparrow_\rho C_1 \quad \Delta; \Xi; \Gamma \vdash s_3 \Uparrow_\rho C_3 \\ \Delta; \Xi; \Gamma \vdash s_1 : \mathit{List}\ i_1 \quad \Delta; \Xi; \Gamma \vdash s_3 : \tau\ i_3 \\ [\![i_1]\!]\rho = v_1 \quad [\![i_3]\!]\rho = v_3 \\ [\![i_2]\!](v_3, v_1, \rho) = v' \\ \Delta; \Xi; \Gamma \vdash s_2 \Uparrow_\rho^{v_1, v_3} v', C'\end{array}}{\Delta; \Xi; \Gamma \vdash \text{elimlist}\ s_1\ (x, \mathit{acc} \mapsto s_2)\ s_3 \Uparrow_\rho C_1 + C_3 + C'}$$

Figure 5.2: Statement cost by execution trace

$$\text{ListCost}\ \frac{\begin{array}{c}\Delta;\Xi;\Gamma \vdash s_1 \Uparrow_\rho C_1 \quad \Delta;\Xi;\Gamma \vdash s_3 \Uparrow_\rho C_3 \\ (j, Nat), (js, List), \Delta; (i_1 \approx j :: js), \Xi; (x, Nat, j), (xs, List, js), \Gamma \vdash s_2 \Uparrow_{v_1,\rho} C_2 \\ \Delta;\Xi;\Gamma \vdash s_1 : List\ i_1 \quad [\![i_1]\!]\rho = v_1 \\ \Delta;\Xi \vdash i_1 \rightsquigarrow_\rho (j :: js) \quad \Delta;\Xi \vdash i_1 \rightsquigarrow_\rho [] \\ \text{IfNil}(v_1, C_2, C_3, C')\end{array}}{\Delta;\Xi;\Gamma \vdash \text{caselist}\ s_1\ ((x :: xs) \Rightarrow s_2);\ ([] \Rightarrow s_3) \Uparrow_\rho C_1 + C'}$$

$$\text{ConsCost}\ \frac{\begin{array}{c}\Delta;\Xi;\Gamma \vdash s_1 \Uparrow_\rho C_1 \\ \Delta;\Xi;\Gamma \vdash s_1 : List\ i_1 \\ [\![i_1]\!]\rho = w :: ws \\ \Delta;\Xi \vdash i_1 \not\rightsquigarrow_\rho [] \\ (j, Nat), (js, List), \Delta; (i_1 \approx j :: js), \Xi; (x, Nat, j), (xs, List, js), \Gamma \vdash s_2 \Uparrow_{w::ws,\rho} C_2\end{array}}{\Delta;\Xi;\Gamma \vdash \text{caselist}\ s_1\ ((x :: xs) \Rightarrow s_2);\ ([] \Rightarrow s_3) \Uparrow_\rho C_1 + C_2}$$

$$\text{NilCost}\ \frac{\begin{array}{c}\Delta;\Xi;\Gamma \vdash s_3 \Uparrow_\rho C_3 \\ \Delta;\Xi;\Gamma \vdash s_1 : List\ i_1 \\ \Delta;\Xi \vdash i_1 \not\rightsquigarrow_\rho (j :: js)\end{array}}{\Delta;\Xi;\Gamma \vdash \text{caselist}\ s_1\ ((x :: xs) \Rightarrow s_2);\ ([] \Rightarrow s_3) \Uparrow_\rho C_3}$$

Figure 5.2: Statement cost by execution trace (continued)

NilCost corresponds to the case of a list known to be nil. In this case the scrutinee can be thrown away without evaluating since we will simply discover that it's nil, gaining no new information. Thus the cost of the case will be the cost of evaluating the nil branch s_3, i.e. the cost will be C_3.

ConsCost corresponds to the case of a list known to be non-nil. In this case the scrutinee *must be evaluated* since the head and tail must be bound to x and xs respectively. Since the scrutinee s_1 could be a complicated expression it might take some computation in order to gain access to the list's head and tail; thus the cost of breaking this expression down is given by C_1. Finally the cons case of the branch—having cost C_2—must be evaluated. Thus the total cost of this expressions is the sum of these two costs: $C_1 + C_2$.

ListCost corresponds to the final case: a term of unknown shape. Since nothing is known about the input, the cost is "stuck" with a conditional. As was the case in the cons case, first the scrutinee must be evaluated,

having cost C_1. Next, having broken down s_1 into a list value, we must inspect it and decide whether the nil or non-nil branch should be evaluated which is the responsibility of IfNil defined in appendix F on page 209. Since either of the branches might be taken, the cost of both branches C_1 and C_2 are required. Finally the IfNil relation is given by cases on v_1, binding C' to C_2 when v_1 is non-nil, and to C_3 when v_1 is nil. Thus the total cost of the case is the cost of evaluating scrutinee, plus the cost of the branch taken: $C_1 + C'$.

The final rules ElimNatCost and ElimListCost correspond to the costing of loops over numbers and lists respectively. Note that because the cost of the body of the loop can depend on the shape of its inputs x and xs there is no closed form solution that would suitably capture the cost of repeatedly executing a general loop body. However in special cases such as a constant cost body, closed form solutions such as taking the product of the body cost are possible. Thus to cost a loop, it must be completely evaluated and hence its input must be known. Note that this is a stronger requirement than a simple loop bound since in the case of a list, every member's value must be expressible. Fortunately such static information is available courtesy of the input's semantic index.

First consider the rule ElimNatCost, the total cost appeals to the loop body relation $\Delta; \Xi; \Gamma \vdash s \Uparrow_{\rho}^{v_1, v_3} v', C'$ defined in figure 5.3. This relation accepts an item v_1 to loop over and an "accumulator seed" v_3 which is used to evaluate the loop's base case. However these values are also used to extend the context because the term s cannot be evaluated in ρ alone. This is because s is only well-formed in the contexts Δ and Γ extended with x and xs, allowing access to the current input and accumulator values.

The loop body relation in figure 5.3 breaks the cost calculation into the base case DoneNatCost and recursive case StepNatCost of the loop trace. Before going over the details it is helpful to recall the traditional fold function, or eliminator, on natural numbers.

$$\begin{aligned} \mathit{foldnat} &: (\mathbb{N} \to A \to A) \to A \to \mathbb{N} \to A \\ \mathit{foldnat}\ f\ v\ 0 &= v \\ \mathit{foldnat}\ f\ v\ (n+1) &= f\ n\ (\mathit{foldnat}\ f\ v\ n) \end{aligned}$$

The above pattern will be followed by the DoneNatCost and the StepNatCost rules, corresponding to the 0 and $n + 1$ cases respectively. Intuitively the

base case returns the seed v without doing any work, thus having zero cost. The recursive case will calculate a recursive cost and perform one more step of work, summing the two costs and returning the newly accumulated value.

As in *foldnat*, the base case does no work, and simply returns the initial accumulator value v. Thus the resulting accumulator is v and the total cost is 0. Costing a recursive step of loop evaluation is where the actual work occurs. Intuitively the overall cost will be the cost of evaluating the rest of the loop, plus the cost of evaluating the loop body for the current step. Thus the StepNatCost rule first computes the cost of the rest of the loop $\Delta; \Xi; \Gamma \vdash s \Uparrow_{\rho}^{n,v} v', C$ feeding the resulting accumulator value v' into the evaluator $[\![j]\!](n+1, v', \rho) = v''$ to compute the new accumulator value v'' required by the next round of the loop. Finally the cost of the loop body for this iteration is found to be C' by $\Delta; \Xi; \Gamma \vdash s \Uparrow_{n+1,v'',\rho} C'$. Thus the overall cost of the loop for $n+1$ iterations is the sum of the costs of performing n iterations plus this iteration's body i.e. $C + C'$.

The last remaining rule, ElimListCost, is nearly identical to the rule ElimNatCost because lists may only contain natural numbers. Thus iterating over a list repeatedly yields the list head—a number—along with the current accumulator. Similarly, iterating over a natural number counts down, repeatedly yielding the current number along with the current accumulator. Thus as was the case with ElimNatCost the total cost of eliminating a list is the cost of evaluating the initial accumulator C_3, plus the cost of evaluating the list to eliminate C_1, plus sum of the costs of the loop body for each round of iteration C' i.e. $C_1 + C_3 + C'$. As was the case in ElimNatCost, ElimListCost relies on the loop body relation defined in figure 5.3 to compute the total cost of the loop. The two rules DoneListCost and StepListCost follow the same pattern as DoneNatCost and StepNatCost, except that iteration is performed on the tail of the list rather than counting down a number.

Using this relation it's trivial to give a precise cost for the head function from chapter 3 noting it has the same representation in both $\mathscr{L}$ and $\mathscr{S}$.

$$\mathit{head}\ \mathit{xs} = \text{caselist}\ \mathit{xs}\ ((x :: \mathit{xs}') \Rightarrow x);\ ([] \Rightarrow \text{dead}\ (i :: \mathit{is})\ [])$$

Costing the above is a single application of the ConsCost rule which clearly holds since [] is unreachable—proof of which follows from the application of

$\boxed{\Delta; \Gamma \vdash s \Uparrow_\rho^{\mathcal{V},\mathcal{V}} C}$ (traced iteration cost)

$$\text{DoneNatCost}\ \frac{(i, Nat), (acc, \tau), \Delta; \Xi; (x, Nat, i), (xs, \tau, acc), \Gamma \vdash s : \tau\ j}{\Delta; \Xi; \Gamma \vdash s \Uparrow_\rho^{0,v} v, 0}$$

$$\text{StepNatCost}\ \frac{\begin{array}{c}(i, Nat), (acc, \tau), \Delta; \Xi; (x, Nat, i), (xs, \tau, acc), \Gamma \vdash s : \tau\ j \\ \Delta; \Xi; \Gamma \vdash s \Uparrow_\rho^{n,v} v', C \\ [\![j]\!](n, v', \rho) = v'' \\ \Delta; \Xi; \Gamma \vdash s \Uparrow_{n,v'',\rho} C'\end{array}}{\Delta; \Xi; \Gamma \vdash s \Uparrow_\rho^{n+1,v} v'', C + C'}$$

$$\text{DoneListCost}\ \frac{(i, Nat), (acc, \tau), \Delta; \Xi; (x, Nat, i), (xs, \tau, acc), \Gamma \vdash s : \tau\ j}{\Delta; \Xi; \Gamma \vdash s \Uparrow_\rho^{[],v} v, 0}$$

$$\text{StepListCost}\ \frac{\begin{array}{c}(i, Nat), (acc, \tau), \Delta; \Xi; (x, Nat, i), (xs, \tau, acc), \Gamma \vdash s : \tau\ j \\ \Delta; \Xi; \Gamma \vdash s \Uparrow_\rho^{vs,v} v', C \\ [\![j]\!](n, v', \rho) = v'' \\ \Delta; \Xi; \Gamma \vdash s \Uparrow_{n,v'',\rho} C'\end{array}}{\Delta; \Xi; \Gamma \vdash s \Uparrow_\rho^{n::vs,v} v'', C + C'}$$

Figure 5.3: Iteration cost by execution trace

dead. Thus the cost is derived as follows.

$$\begin{array}{l}(i, Nat), (is, List), \emptyset; \\ \qquad \emptyset; \vdash \text{caselist } xs \begin{Bmatrix} x :: xs' \Rightarrow x; \\ [] \Rightarrow \text{dead } (i :: is)\ [] \end{Bmatrix} \Uparrow_{w::ws,\emptyset} h^\star(xs) + h^\star(x) \\ (xs, List, i :: is), \emptyset\end{array}$$

Given that the cost of the body is.

$$\frac{C = h^\star(x)}{\begin{array}{r}(i, Nat), (is, List), \emptyset; \\ (i :: is \sim j :: js), \emptyset; \vdash x \Uparrow_{v,vs,w::ws,\emptyset} h^\star(x) \\ (x, Nat, j), (xs', List, js), (xs, List, i :: is), \emptyset\end{array}}$$

Along with the cost of evaluating the scrutinee, required to gain access to

its head and tail.

$$\frac{C = h^\star(xs)}{\begin{array}{l}(i, Nat), (is, List), \emptyset; \\ \qquad\qquad \emptyset; \vdash xs \Uparrow_{w::ws,\emptyset} h^\star(xs) \\ (xs, List, i :: is), \emptyset\end{array}}$$

While these derivations are unwieldy to write by hand, Idris tends to infer many of the terms, including the context and environment. Unfortunately this is double edged—inference of these terms is costly and checking even a small term such as a single loop takes minutes on a modern Haswell machine in addition to upwards of 5GBs of RAM. For instance Idris can be asked to verify that the cost of a loop over 2 is the sum of repeatedly evaluating the body plus the cost of evaluating the initial accumulator value and the value to iterate over. This proposition is given by

```
CostTrace []
          []
          (ElimNat (Atomic (NLit 2)) "i'" "acc"
                   (Atomic (Add (Var Here) (Var (There Here))))
                   (Atomic (NLit 1)))
          (costAExp (NLit 2) +
           (costAExp (Add (Var Here) (Var (There Here))) +
            (costAExp (Add (Var Here) (Var (There Here))) +
             (costAExp (Add (Var Here) (Var (There Here)))))) +
           (costAExp (NLit 1)))
```

To prove this, Idris need only be supplied with the high-level constructors ElimNatCost, AtomicCost, AtomicCost and the rest can be inferred. The exact proof is

```
ElimNatCost ACost
            (SucCost ACost (SucCost ACost (ZeroCost ACost)))
            ACost
```

A key advantage of this cost strategy using an EDSL in a dependently typed context is that the cost of a term can be reasoned about directly in the host language. Importantly it's possible to give a mechanical, correct-by-construction guarantee that these costs are over the correct program, and not one which has been mistranslated. Specifically the trace based cost relation allows for reasoning about statements in $\mathscr{S}$; however this is not the language the program's are written in and thus in the next section we discuss a means of translating from $\mathscr{L}$ to $\mathscr{S}$ automatically. As earlier noted,

the statement language is meant to capture the rough structural properties of the code to be executed.

5.2 Decomposing semantics indexed expressions

This decomposition will be *index expression preserving*, and therefore correct by construction. It will be shown this is the case by considering that if the statement language is given a index expression respecting semantics, by the canonicity of values any transformation which is index expression respecting is automatically semantics preserving.

Although a means of costing programs has been developed, the analysis is over a newly proposed statement language rather than the expression language $\mathscr{L}$. Rather than adapting the analysis, a decomposition of index expression expressions into the statement language will be developed. As stated, the correctness of the decomposition will be guaranteed by ensuring it is *index expression preserving* i.e. by indexing statements by the index expression of the concrete expression whose structure is described. The guarantee of the transformation's correctness is not immediately obvious. However note that index expressions can be viewed not only as a restriction of an expression's or a statement's shape but also as capturing some semantic invariance. In this way the expression and statement types become *views* [MM04b] on a index expression. Thus if a transformation is index expression preserving, it is semantics preserving. To see roughly why, consider that two index expression respecting notions of evaluation produce values at the same index expression index; however by the canonicity of indexed values given in theorem 3.5.3 on page 100 this implies that the indexed values themselves are too equivalent. Thus showing the equivalence between any transformation on a syntax tree is reduced to showing that it is index expression preserving. As will be seen later, the statement language will not only lend itself to static analysis, but also simplify the process of performing compilation to reasonably efficient object code.

First the syntax of indexed, atomically costed expressions should be captured. Recall that such expressions include variables, values, and the "builtin-operation" of numeric addition. The notions of variables and list values can remain unchanged; however numeric literals are no longer expressed

as a chain of successors on zero. Due to the lack of a builtin successor operation, the handling of numbers requires that towers of successor be collapsed to a single numeric literal. In the case of handling built-in operations there is a choice: some languages such as Agda allow explicit tagging of functions as built-in so that semantically they are treated using their definition, but at compile time result in the emission of efficient code. In our case anything with a index expression that's semantically equivalent to addition could be used. Such a rule could be enforced with a trick similar to the rewrite expression. However in the interest of simplicity only terms following a particular shape will be translated to primitive addition. In chapter 4 on page 108 addition will be given as a macro in $\mathscr{L}$ that will always result in the extraction of efficient code. In particular because there is no automated means for testing function equivalence such a flexible rule would only be suitable for less-automated structural decompositions that heavily involve the programmer.

The rules for atomically costed expressions are given in figure 5.4 The rules use a pre-ordained definition for addition on index expressions. Recall that the goal is to lift structural elements out of the expression language from chapter 3, thus the remaining language constructors from $\mathscr{L}$ will map to the structurally costed portion of $\mathscr{S}$. Given the great similarity between the two languages, only two new syntactic elements need to be introduced: let binding and atomic expression lifting. These elements are required because like in C, addition expression can no longer have expressive control flow elements such as loops and branches embedded in them.

Given a notion of well-formedness it's possible to embed $\mathscr{S}$ as an EDSL into Idris using the usual type indexing trick to ensure correctness. The translation of $\mathscr{S}$ syntax into an EDSL using Idris is relatively straightforward. First consider atomic expressions. Literals and variables will take their usual shape. In the case of addition, as was noted although it's possible to give a flexible definition allowing any index expression which is semantically equivalent to the usual definition of addition, its not clear that much value is added given the additional complexity, thus the index is taken to be the standard definition of addition as a loop over the augend.

```
data AExp : Ctx ictx -> (t : Ty) -> Ix ictx t -> Type where
  Add : AExp ctx TyNat i1 -> AExp ctx TyNat i2
     -> AExp ctx TyNat (i1 `plus` i2)
  NLit : (n : Nat) -> AExp ctx TyNat (nat n)
  Var : (elt : Elem (x, (t ** i)) ctx) -> AExp ctx t i
```

$\boxed{\Delta;\ \Xi;\ \Gamma \vdash \text{s} :\ \tau\ \ \text{S}}$ (well-formed, indexed statements)

$$\frac{\Gamma \ni_v (x, \tau, i)}{\Delta; \Xi; \Gamma \vdash v : \tau\ i} \qquad \frac{}{\Delta; \Xi; \Gamma \vdash n : Nat\ n}$$

$$\text{Add}\frac{\Delta; \Gamma \vdash a_1 : Nat\ i_1 \quad \Delta; \Gamma \vdash a_2 : Nat\ i_2}{\Delta; \Gamma \vdash a_1 + a_2 : Nat\ (plus\ i_1\ i_2)}$$

$$\text{Rewrite}\frac{\Delta; \Gamma \vdash s : \tau\ i}{\Delta; \Gamma \vdash \text{rewrite}\ j\ p\ s : \tau\ j}$$

$$\frac{}{\Delta; \Xi; \Gamma \vdash [] : List\ []} \qquad \frac{\Delta; \Xi; \Gamma \vdash s_1 : Nat\ i_1 \quad \Delta; \Xi; \Gamma \vdash s_2 : List\ i_2}{\Delta; \Xi; \Gamma \vdash s_1 :: s_2 : List\ i_1 :: i_2}$$

$$\frac{\Delta; \Xi; \Gamma \vdash s_1 : \tau_1\ i_1 \quad \Delta; \Xi; (x, \tau_1, i_1), \Gamma \vdash s_2 : \tau_2\ i_2}{\Delta; \Xi; \Gamma \vdash \text{let}\ x = s_1\ \text{in}\ s_2 : \tau\ i_2}$$

$$\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash s_1 : Nat\ i_1 \\ (j, Nat), (js, List), \Delta; (i_1 \approx j :: js), \Xi; (x, Nat, j), (xs, List, js), \Gamma \vdash s_2 : \tau\ i_2 \\ \Delta; \Xi; \Gamma \vdash s_3 : \tau\ i_3\end{array}}{\Delta; \Xi; \Gamma \vdash \text{caselist}\ s_1\ ((x :: xs) \Rightarrow s_2);\ ([] \Rightarrow s_3) : caselist\ i_1\ ((j :: js) \Rightarrow i_2);\ ([] \Rightarrow i_3)}$$

$$\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash s_1 : Nat\ i_1 \\ (j, Nat), (acc, \tau), \Delta; \Xi; (x, Nat, j), (xs, \tau, acc), \Gamma \vdash s_2 : \tau\ i_2 \\ \Delta; \Xi; \Gamma \vdash s_3 : \tau\ i_3\end{array}}{\Delta; \Xi; \Gamma \vdash \text{elimlist}\ s_1\ (x, xs \mapsto s_2)\ s_3 : elimlist\ i_1\ (j, acc \mapsto i_2)\ i_3}$$

$$\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash s_1 : Nat\ i_1 \\ (j, Nat), (js, List), \Delta; \Xi; (x, Nat, j), (xs, \tau, acc), \Gamma \vdash s_2 : \tau\ i_2 \\ \Delta; \Xi; \Gamma \vdash s_3 : \tau\ i_3\end{array}}{\Delta; \Xi; \Gamma \vdash \text{elimnat}\ s_1\ (x, xs \mapsto s_2)\ s_3 : elimnat\ i_1\ (j, acc \mapsto i_2)\ i_3}$$

Figure 5.4: Typing rules for statements and atomic expressions

The encoding of statements is close enough to expressions that only the special cases of atomic expressions and let statements are included, the full translation is given in appendix D on page 199.

```
data Stmt : IxConstraints ictx -> Ctx ictx -> (t : Ty)
          -> Ix ictx t -> Type where
  Atomic : AExp ctx t i -> Stmt facts ctx t i
  Let : (x : Name)
     -> {ctx : Ctx ictx}
     -> Stmt facts ctx t1 i1
     -> Stmt facts ((x, (t1 ** i1)) :: ctx) t2 i2
     -> Stmt facts ctx t2 i2
```

The atomic expressions leave little to the imagination, blindly lifting a well-formed expression into the statement language, preserving its index expression and type. However the handling of let is slightly more interesting in its remarkable simplicity. Since the introduction of a let binding induces no change in the semantics of a term, only the statement context is modified, introducing the newly bound variable `x` along with its index expression `i1` and type `t1`. Thus ensuring the sound introduction of binding is deferred to the algorithm decomposing an expression in $\mathscr{L}$ into a statement in $\mathscr{S}$.

Fortunately the function for decomposing an expression in $\mathscr{L}$ to one in $\mathscr{S}$ is relatively simple thanks to the use of dependent types. In fact, as will be seen the rules are so strict that for example attempting to misplace a variable when decomposing an addition operation will result in a type-checking error because the resulting index expression would have been rendered incorrect.

Intuitively expression decomposition will work by repeatedly transforming sub-expressions and producing an auxiliary proof that the resulting index expression is equivalent to the input index expression. Finally at the top level we will arrive at a statement without rewrites in some new index expression j along with a proof ofindex expression equality between the input index expression i and the new index expression j. To carry this proof of transporting the input index expression i to the a statement with an equivalent output index expression j, a type of statements with a top-level rewrite is introduced.

```
data StmtRewritePath : IxConstraints ictx
                    -> Ctx ictx
                    -> (t : Ty)
                    -> Ix ictx t
                    -> Ix ictx t
                    -> Type where
```

```
MkRewrite : ((ixEnv : IxEnv ictx) ->
              EvalConstraints ixEnv facts ->
              eval i ixEnv = eval j ixEnv)
         -> Stmt facts ctx t j
         -> StmtRewritePath facts ctx t i j
```

Note that this precisely follows the structure of the rewrite rule in section 3.4.2 on page 88. Of course we wish to talk about a index expression respecting statement without particular care for the shape of j and thus the new index is best hidden with an existential.

```
StmtRewrite : IxConstraints ictx -> Ctx ictx -> (t : Ty)
           -> Ix ictx t -> Type
StmtRewrite {ictx=ictx} facts ctx t i =
  (j : Ix ictx t ** StmtRewritePath facts ctx t i j)
```

Given a means of flexibly building correct-by-construction statements respecting some "source" index expression i, it's possible to give a mechanization of expression decomposition.

```
decomposeRewrite : Exp facts ctx t i
                -> StmtRewrite facts ctx t i
```

As a warm-up handling variables and lists is trivial because the representation is identical in $\mathscr{L}$ and $\mathscr{S}$, thus the only work required is in the construction of the index expression equivalence proof. For variables the equivalence is trivial since no transformation is performed.

```
decomposeRewrite (Var {x=x}{t=t}{i=i} elt) =
  (i ** MkRewrite (\ixEnv, knowns => Refl)
                  (Atomic (Var elt)))
```

However in the case of statements the head and tail must be recursively transformed. These sub-trees may have rewrites, additions, successor, etc. and thus will their transformation produces both a statement and a rewrite. In order to compose these two, their rewrites must also be composed.

```
decomposeRewrite [] =
  ([] ** MkRewrite (\ixEnv, knowns => Refl) [])
decomposeRewrite (e1 :: e2) = case decomposeRewrite e1 of
  (j1 ** MkRewrite j1Ei1 s1) => case decomposeRewrite e2 of
    (j2 ** MkRewrite j2Ei2 s2) =>
      (j1 :: j2 **
       MkRewrite (\ixEnv, knowns => ?j1j2Ok) (s1 :: s2))
```

As stated, `e1` of index expression `i1` is decomposed into the statement `s1` whose new index expression is `j1`. However we are equipped with a proof

`j1Ei1` that `j1` is semantically equivalent to the index expression `i1`, i.e. that `eval i1 ixEnv1 = eval j1 ixEnv1` in the current constraint context `facts`. Decomposing the tail `e2` into `s2` of index expression `j2` produces a similar semantic correctness proof. Given this knowledge, what is the goal `?j1j2Ok`?

Since the top-level expression `e1 :: e2` is being deconstructed, the required proof states that the final output index expression `j1 :: j2` is semantically equivalent to the input index expression `i` which of course is known to be `i1 :: i2`. Thus the goal reduces as follows.

```
  eval (i1 :: i2) ixEnv
= eval i1 ixEnv :: eval i2 ixEnv
= eval j1 ixEnv :: eval j2 ixEnv
```

Of course by assumption the above head and tail are equivalent, thus by repeated application of rewriting the obligation is satisfied.

```
?j1j2Ok = rewrite j2Ei2 ixEnv knowns in
          rewrite j1Ei1 ixEnv knowns in
            Refl
```

The handling of numbers is a particularly interesting case because in $\mathscr{L}$ they are represented as the repeated application of successor to zero. Ideally these towers of constructors should be optimized away and thus collapsed into a single literal. The handling of 0 is trivial, simply return a new 0 literal.

```
decomposeRewrite Z = (Z ** MkRewrite (\ixEnv, knowns => Refl)
                                     (Atomic (NLit Z)))
```

However handling successor is more interesting and depending on the desired level of optimization, can be handled in several distinct cases. Consider the successor input `S e`, after first decomposing `e` into $\mathscr{S}$ how should decomposition proceed?

```
decomposeRewrite (S e) = case decomposeRewrite e of
```

The simplest case is when `e` happened to be another application of successor or zero and thus is decomposed into a numeric literal. In this case the current application of successor can simply be absorbed into the literal.

```
(nat n ** MkRewrite jEi (Atomic (NLit n))) =>
  (nat (S n) **
   MkRewrite (\ixEnv, knowns => ?sucOk)
             (Atomic (NLit (S n))))
```

As usual proof that this is a legal, index expression respecting operation must be given. Fortunately the decomposition of `e` provides a proof `jEi` that the index expression `i` of `e` is the constant `nat n`. Thus by an application of the congruence rule `S i = S (nat n) = nat (S n)` as required.

```
?sucOk = cong (jEi ixEnv knowns)
```

On the opposite end of the spectrum, given a non-atomic statement nothing can be done. Thus the resulting value must be extracted with a let binding so that it can be incremented using an addition instruction.

```
(j ** MkRewrite jEi s) =>
  (S Z `plus` j **
   MkRewrite (\ixEnv, knowns => rewrite jEi ixEnv knowns in
                                  plusOneIsSuc ixEnv)
             (Let "sucS" s (Atomic (Add (NLit 1)
                                    (Var Here)))))
```

The proof that this transformation is index expression respecting follows from the fact that adding one to a term by looping over it and replacing zero with one is equal to simply applying the successor constructor. The term `jEi` proves that `eval j ixEnv = eval i ixEnv` however the index expression of the let body is the aforementioned loop over `j`. Thus the goal is reduced to proving this equality. Fortuntely Idris can deduce `plusOneIsSuc` by simply reducing both sides.

```
plusOneIsSuc : {j : Ix ictx TyNat}
            -> (ixEnv : IxEnv ictx)
            -> S (eval j ixEnv) =
               natFold (\n, vacc => S vacc) (eval j ixEnv)
                       (S Z)
plusOneIsSuc ixEnv = Refl
```

The final case to handle is when `e` is decomposed into a non-literal atomic expression e.g. a variable. In this instance, like the case of a statement no optimization can be done, and thus the term should be incremented with an add instruction.

```
(j ** MkRewrite jEi (Atomic a)) =>
  (S Z `plus` j **
   MkRewrite (\ixEnv, knowns => rewrite jEi ixEnv knowns in
                                  plusOneIsSuc ixEnv)
             (Atomic (Add (NLit 1) a)))
```

The reader may note the handling of two more possible cases could optimize the resulting term when addition is performed over a literal on either side.

Such an optimization is certainly possible and is readily shown to be index expression respecting.

Iteration decomposition is similar for both list and nat, thus only nat iteration is covered. Unsurprisingly the strategy is to directly decompose the body, the expression to iterate over, and the initial value; however proving equivalence of the new index expression is tricky, requiring a proof folding with two step function agreeing everywhere are equal. This proof is given by a lemma `rewriteElimNat`.

```
decomposeRewrite (ElimNat e1 i' acc e2 e3) =
  case decomposeRewrite e1 of
    (j1 ** MkRewrite j1Ei1 s1) => case decomposeRewrite e2 of
      (j2 ** MkRewrite j2Ei2 s2) => case decomposeRewrite e3 of
        (j3 ** MkRewrite j3Ei3 s3) =>
          (ElimNat j1 acc i' j2 j3 **
           MkRewrite (\ixEnv, knowns =>
             rewriteElimNat (j1Ei1 ixEnv knowns)
                            j2Ei2
                            knowns
                            (j3Ei3 ixEnv knowns))
             (ElimNat s1 i' acc s2 s3))
```

The proposition `rewriteElimNat` is considerably more complicated than its proof, asserting that if the evaluation of `j2` and the evaluation of `i2` agree then so do folds where each step evaluates either respective body.

```
rewriteElimNat
  : {w1, w1' : Nat}
 -> {i2, j2 : Ix ((acc, t) :: (i', TyNat) :: ictx) t}
 -> {w3, w3' : EvalTy t}
 -> w1 = w1'
 -> (ixEnv : IxEnv ictx) ->
     EvalConstraints ixEnv
       (weakenConstraints facts
                          [(acc, t)
                          ,(i', TyNat)]) ->
     eval i2 ixEnv = eval j2 ixEnv
 -> EvalConstraints ixEnv facts
 -> w3 = w3'
 -> natFold (\n, vacc => eval i2 (vacc :: n :: ixEnv)) w3 w1 =
    natFold (\n, vacc => eval j2 (vacc :: n :: ixEnv)) w3' w1'
```

The proof of this term is a direct consequence of the fact that whenever two functions are extensionally equal, so too is fold.

```
rewriteElimNat {w1=w1}{w3=w3}{i2=i2}{j2=j2}{ixEnv=ixEnv}
```

```
Refl i2Ej2 knowns Refl =
natFoldFSubst {f = \n, vacc => eval i2 (vacc :: n :: ixEnv)}
              {g = \n, vacc => eval j2 (vacc :: n :: ixEnv)}
              (\n, vacc => i2Ej2 (vacc :: n :: ixEnv)
                                 (weakenKnowns knowns [_,_]))
              w1 w
```

The proof is given in listing D.3 on page 206.

The next case to handle is an explicit coercion by `Rewrite` which is key to proving the rewrite minimality property. Intuitively rewriting a term twice in a row should be collapsable since rewriting is ultimately a statement of propositional equality. This intuition indeed translates to an application of transitivity. First the rewritten term `e` of index expression `i` is reduced to `s` with new index expression `k` along with a proof that `i` and `k` are semantically equivalent. However the goal is to show an equivalence between the new index expression `k` and the index expression `j` of the rewrite itself. This of course is precisely from where the need for transitivity arises.

```
decomposeRewrite (Rewrite j iToJ e) =
  let (k ** MkRewrite iEk s) = decomposeRewrite e
  in (k ** MkRewrite (\ixEnv, knowns =>
                        sym (iToJ ixEnv knowns) `trans`
                        iEk ixEnv knowns)
                     s)
```

Thus all rewrites appear to be lifted to the top scope effectively erasing them. Unfortunately this is not so, as case analysis *introduces* rewrites. To understand why this is necessary consider how decomposition might proceed over a case where both branches are reachable.

```
decomposeRewrite (CaseList {i1=i1} {i2=i2} {i3=i3} e1 hd tl
                    (Reachable (CanReach sub2 i1ToJ) e2)
                    (Reachable (CanReach sub3 i1ToK) e3))
```

Intuitively—but wrongly so—the three expressions could be transformed into statements with new index expressions `j1`, `j2`, and `j3`. Then a index expression `CaseList j1 hd tl j2 j3` might be returned with a new `CaseList` statement, swapping out the three expressions for their decomposed statements.

```
case decomposeRewrite e1 of
  (j1 ** MkRewrite j1Ei1 s1) => case decomposeRewrite e2 of
    (j2 ** MkRewrite j2Ei2 s2) => case decomposeRewrite e3 of
      (j3 ** MkRewrite j3Ei3 s3) =>
```

Unfortunately this fails—infuriatingly, stale copies of the original index expressions `i1`, `i2`, and `i3` remain in the constraint context! Thus the body statements `s2` and `s3` are ill-typed, wrongly generating constraints on their *new* index expressions `j2` and `j3`. The solution is simply to swap out the new index expression for the old using coercions.

```
(CaseList i1 hd tl i2 i3 **
 MkRewrite
   (\ixEnv, knowns => Refl)
   (CaseList (Rewrite (\ixEnv, known =>
                sym (j1Ei1 ixEnv known)) s1)
      hd tl
      (Reachable
         (CanReach sub2 (\ixEnv, known, known' =>
            i1ToJ ixEnv known known'))
         Rewrite {j=i2} (\ixEnv, known =>
           sym (j2Ei2 ixEnv known)) s2))
      (Reachable
         (CanReach sub3 (\ixEnv, known, known' =>
            i1ToK ixEnv known known'))
         (Rewrite {j=i3} (\ixEnv, known =>
            sym (j3Ei3 ixEnv known)) s3))))
```

The remaining cases considering reachability of each branch follow the same strategy and thus are omitted.

Although all the possible shapes have been handled, the application of addition-as-iteration has not yet been optimized into an atomic instruction so the resulting statements still make inefficient use of the underlying hardware. As is common in many languages including Idris and Agda some operations are given special compilation behavior in addition to their user facing semantic definition. Here the "addition loop" given as a fold on the augend `ElimNat e1 i' acc (S (Var Here)) e2` is considered to be the canonical notion of addition and is optimized accordingly, however were functions supported natively—as opposed to deferring to host support—a particular function name could be tagged for optimization.

Similar to the case of handling numeric literals, the handling of addition is broken down into case depending on the shape of the decomposed augend `e1` and addend `e2`. In the simplest case both `e1` and `e2` are atomic expressions and thus their sum is simply an add node over their atomic decompositions `a1` and `a2`.

```
decomposeRewrite (ElimNat e1 i' acc (S (Var Here)) e2) =
  case  decomposeRewrite e1 of
```

```
    (j1 ** MkRewrite j1Ei1 (Atomic a1)) =>
      case decomposeRewrite e2 of
        (j2 ** MkRewrite j2Ei3 (Atomic a2)) =>
          (j1 `plus` j2 **
           MkRewrite (\ixEnv, knowns => ?addOk)
                     (Atomic (Add a1 a2)))
```

Clearly such a transformation is valid because the shape of the index expression does not change, in both cases it is a sum—expressed as a loop—over the inputs; however the inputs `e1` and `e2` have been given new index expressions upon decomposition. Thus the obligation is a case of swapping out equal arguments

```
rewriteAddAtomic : w1 = w1'
                -> w3 = w2'
                -> (f : Nat -> Nat -> Nat)
                -> natFold f w3  w1 = natFold f w2' w1'
rewriteAddAtomic Refl Refl f = Refl
```

As it turns out this rewrite rule is flexible enough to solve all four cases of addition because each reduces to an equality on inputs.

```
?sucOk = rewriteAddAtomic (j1Ei1 ixEnv knowns)
                          (j2Ei3 ixEnv knowns)
                          _
```

The remaining three cases are all similar where either input, or both are decomposed into non-atomic statements and thus must be let-bound before use in an addition node. In the worse case both terms are non-atomic.

```
case decomposeRewrite e1 of
  (j  1 ** MkRewrite j1Ei1 s1) =>
    case decomposeRewrite e2 of
      (j2 ** MkRewrite j2Ei3 s2) =>
        (j1 `plus` j2 **
         MkRewrite (\ixEnv, knowns =>
           rewriteAddAtomic (j1Ei1 ixEnv knowns)
                            (j2Ei3 ixEnv knowns) _)
           (Let "s1add" s1
             (Let "s2add" (weakenStmt s2 [_])
               (Atomic (Add (Var (There Here))
                            (Var Here))))))
```

Note that the second statements `s2` had to undergo explicit weakening since it's defined in a scope without the `s1add` variable. Similarly when one or the other inputs is atomic, the resulting expression is well-defined in a smaller context, without the outer let-bound variable.

```
case decomposeRewrite e2 of
  (j2 ** MkRewrite j2Ei3 (Atomic a2)) =>
    (j1 `plus` j2 **
     MkRewrite (\ixEnv, knowns =>
       rewriteAddAtomic (j1Ei1 ixEnv knowns)
                        (j2Ei3 ixEnv knowns) _)
       (Let "s1add" s1
            (Atomic (Add (Var Here) (weaken a2 [_])))))
```

The final case is symmetric and thus omitted.

Thus the function `decomposeRewrite` gives a formally verified means of extracting low-level, machine supported features in a correct-by-construction way. We have also shown how to efficiently handle machine supported data types as is done in e.g. Epigram [Bra05b] in a safe, mechanized fashion.

In a rough sense an index expression i can be thought of as some type of abstract interpretation of some concrete statement $s : \tau\ i$ inhabiting it. The statement s communicates how to *efficiently* compute a value, while i communicates some semantic contract that s must meet. Of course unlike in the case of abstract interpretation which usually involves some rough approximation, the resulting values from reducing i and s coincide *precisely*. Specifically Cousot and Cousot formalize a term's abstract interpretation [CC77] as an *approximation* of that term along with a formal connection between the two. For instance we might like to approximate some arithmetic formula $m + n$ in the case we don't know the specific values of m and n. In this case, rather than taking m and n to be natural numbers, they instead can be considered to be ranges if e.g. m and n have some known bounds. In this case $m + n$ is approximated by producing not a specific sum, but by perhaps an inclusive *range* whose lower and upper bound is the sum of the known lower and upper bounds of m and n.

The next section explores precisely how to compile these decomposed expressions directly to C via a low-level, virtual machine. This automated transformation is not only a required step in the full compilation stack, but also exemplifies how the native features exposed by the host can be used to reason about programs. In particular programs can now be precisely costed. It would also be possible to prove upper bounds about their cost behavior using all the standard programming and proving features afforded by Idris without any extra work on the implementor's part as was explored in chapter chapter 4 on page 108.

5.3 Compilation with dead-code elimination

One of the advantages of a deeply embedded framework is the simplicity of performing transformations on data. A claimed advantage of the index expression system developed in chapter 3 on page 66 is the explicitness of unreachable code branches. However thus far they have had little use. In fact, the semantic index language carries no intrinsic reachability information, and were a type of bottom value to be included, the `eval` function would become unimplementable, requiring information be fed into it by the `evalExp` function. This section develops a compiler which uses the case branch reachability information to perform dead code elimination without relying on sophisticated data and control flow analysis. Additionally all well-formedness conditions are internalized yielding a correct-by-construction proof that the transformation is type-respecting, and always produces well-formed terms. The compiler will operate on the `Stmt` language rather than `Exp` obviating the need to extract high-level operations on peano numbers in order to be efficiently handled in hardware. Such a system could be used as an intermediate language, where static analysis might operate on terms without reachability annotations and emitting an `Exp` term to produce a correct-by-construction proof of the dead-code elimination passes' correctness.

The development of the compiler proceeds in parts

1. A description of the abstract machine instructions and data are given.
2. A continuation passing compilation function, verified correct in type but not in semantic index is developed.
3. A printer in the style of Hughes [Hug95] and Wadler [Wad98] is developed emitting compilable "low-level" C.

5.3.1 Describing the abstract machine

So far the correctness of typing and scoping in the languages developed have been ensured by explicit context tracking. Recall that the first order nature of explicit contexts is convenient because it allows for flexible evaluation and reduction which is a necessary feature for index expression rewriting and reachability checking. Since the primary use of the abstract machine developed here is direct compilation, the assembly will use PHOAS which is much more efficiently checked by Idris and avoids tedious context manipulations.

Given that the source language has structured iterators and case analysis the machine will need to support looping and branching. Additionally support is included for basic arithmetic which can be expected of all microprocessors. As is the case in LLVM [LA04], the machine uses static single assignment (SSA) [AWZ88] with an infinite number of registers. However parameterized jumps are used instead of using ϕ-nodes which have a bizarre evaluation depending on the path taken to arrive at the instruction. As observed by Kelsey [Kel95] this is a more natural fit for a functional setting however ultimately either would be suitable since they are equivalent. The equivalence between these two possible design decisions was first explored by Kelsey [Kel95] and further by Appel [App98] and Chakravaty [CKZ03]. Finally since binders are encoded with PHOAS, result values from machine instruction are most naturally captured with continuations in the style of Appel [App92], Kennedy [Ken07], Flanagan et al. [Fla+93], etc..

The machine description is broken into two parts: first a description of the machine's data representation must be given; following this a declaration of the machine's instruction set is possible. Finally the compilation function will be presented in continuation passing style.

A description of machine data

Instructions in the machine naturally operate over registers and values; however structured values such as lists are not directly accessible and must be manipulated through pointers. Thus a machine values is either a numeric literal or a register encoded as a PHOAS variable.

```
MCtx : Type
MCtx = (t : Ty) -> Type

data MLit : Ty -> Type where
  NLit : Nat -> MLit TyNat

data MVal : MCtx -> Ty -> Type where
  Var : mctx t -> MVal mctx t
  Lit : MLit t -> MVal mctx t
```

In order to recapture list destructuring the machine exposes primitive operations for accessing a list's "positions" inspired by the universe of strictly positive types or "container types" [AAG05]; [MAG07]; [Alt+15] and universe of data descriptions [McB10a]. Ghani et al. observed that in a dependently typed setting structured data types can be viewed as a

tree where each node optionally contains some data along with a series of branches which are determined by the structure of the node itself. This structure is termed a *shape*. This idea maps quite well onto machine data where a structure's members can only be accessed by chasing pointers. Moreover the validity of performing member accesses is determined by a value's *shape* which in the realm of runtime systems is more often referred to as a *tag* of a tagged union. For instance a list is either nil or cons and must be tagged as such. In the case of a nil value, no positions are available and the tag is purely discriminative. However in the case of cons two positions are available: head and tail. Thus a type of tags is defined indexed by the type of the indicated constructors. Additionally a tag indexed type of positions is defined, mapping a position in each tag to its type output type.

```
data ConsTag : Ty -> Type where
  TagListNil : ConsTag TyList
  TagListCons : ConsTag TyList

data ConsPos : ConsTag t -> Ty -> Type where
  ConsPosHead : ConsPos TagListCons TyNat
  ConsPosTail : ConsPos TagListCons TyList
```

Although support for $\mathscr{S}$ types is hard-coded, a strategy such as the finite tag enumeration approach of Chapman et al. [Cha+10] could be adapted to provide more general support at the expense of CPU time and error messages.

Given this sufficiently low-level description of the machine's data representation, it's possible to define the instructions that will be needed by the compilation function.

Defining the instruction set

Recall that to support the source language $\mathscr{S}$ only jumps, arithmetic, data allocation, and a few specialized pointer dereferencing instructions are required. Specifically jump if zero, decrement, add, block labeling, and unconditional jump. Backwards jumps to labels are modeled using the PHOAS inspired back-edge representation of Oliviera et al. [SC12]. A label declaration opens a new scope with a PHOAS function (in a distinct namespace from variables) that can be invoked by a jump instruction. As noted rather than using ϕ nodes, data is passed into blocks by passing arguments directly to labels which therefore must be given a type signature. The type of a label in n-parameters is

```
ParamTypes : Nat -> Type
ParamTypes n = Vect n Ty

LabelCtx : Type
LabelCtx = (n : Nat) -> ParamTypes n -> Ty -> Type
```

Thus `LabelCtx` models the type of abstract, typed n-ary arrows. Since a label can only be called with all its arguments passed simultaneously we additionally define the type of machine valued arguments

```
Args : MCtx -> ParamTypes n -> Type
Args mctx ts = All (MVal mctx) ts
```

Similar to the all modality of lists, `All` on vectors effectively maps the proposition `MVal mctx` over `ts` yielding the n length vector of machine values with each type drawn from the next available value in `ts`.

The above non-data instructions can now be rigorously defined in Idris

```
data Asm : MCtx -> LabelCtx -> Ty -> Type where
  Jz : MVal mctx TyNat
    -> Asm mctx lctx t
    -> Asm mctx lctx t
    -> Asm mctx lctx t
  Dec : MVal mctx TyNat
     -> (MVal mctx TyNat -> Asm mctx lctx t)
     -> Asm mctx lctx t
  Add : MVal mctx TyNat
     -> MVal mctx TyNat
     -> (MVal mctx TyNat -> Asm mctx lctx t')
     -> Asm mctx lctx t'
  PJmp : {lctx : LabelCtx} -> {ts : ParamTypes n}
      -> lctx n ts t -> Args mctx ts
      -> Asm mctx lctx t
  PLabel : {lctx : LabelCtx}
        -> (n : Nat)
        -> (ts : ParamTypes n)
        -> (t : Ty)
        -> (vs0 : Args mctx ts)
        -> (lctx n ts t -> All mctx ts -> Asm mctx lctx t)
        -> Asm mctx lctx t
```

While most of these instructions follow standard continuation style, to reiterate the `PLabel` declaration creates a new block in n parameters with initial values drawn from `vs0`. The block is then opened with the newly created label of type `lctx n ts t` along with the n PHOAS encoded variables with types drawn from `ts`: `All mctx ts`.

In addition to the above instructions several specialized instructions for fetching a constructor position, fetching a constructor tag, and atomically allocating and initializing a data type are available. While the first two of these instructions readily map to instructions such as `lea` on amd64, and `getelementptr` on LLVM, the allocation instruction does not requiring an allocation and several move operations. It is argued that the added complexity is a reasonable trade-off to avoid necessitating a complicated model of memory such as separation logic [Rey02] when reasoning about correctness. Tags are encoded at the machine level using integers thus given some pointer type `t` verified by a proof of `ConsTy t`, a program can request the tag of a structured data pointer

```
GetTag : mctx t
      -> {auto isConsTy : ConsTy t}
      -> (MVal mctx TyNat -> Asm mctx lctx t')
      -> Asm mctx lctx t'
```

Given a tag a member of the data structure can be requested by its position

```
GetConsPos : mctx t
          -> {tag : ConsTag t}
          -> ConsPos tag t'
          -> (MVal mctx t' -> Asm mctx lctx t'')
          -> Asm mctx lctx t''
```

Note that the type `t'` to continue in of the value requested is fully described by the type of its position. Finally a structured data type can be allocated and atomically initialized

```
AllocTagged : ConsTy t
           -> (tag : ConsTag t)
           -> ((posTy : Ty) ->
               ConsPos tag posTy ->
               MVal mctx posTy)
           -> (mctx t -> Asm mctx lctx t')
           -> Asm mctx lctx t'
```

The description of initial values are packed into the function of type

```
(posTy : Ty) -> ConsPos tag posTy -> MVal mctx posTy
```

Whereby for every valid position available to the tag, an initializing value of the proper type `posTy` must be produced.

The machine described strikes a good balance between correctness and simplicity ensuring basics such as proper scoping and typing, in addition to some checks on structured data while not requiring values of all data be

known e.g. the unbounded tag values. Notably there is no index expression correctness property thus the correctness of translating from $\mathscr{S}$ is not guaranteed in the usual correct-by-construction style. In the next section this unverified translation will be given in continuation passing style.

5.3.2 Compiling $\mathscr{L}$ with continuations

As has been repeatedly apparent, a great advantage of encoding EDSL's with dependent types is the highly restricted scope of valid programs. By immediately ruling out incorrect possibilities the compiler is often able to automatically decipher desired program structures. This is additionally the case when performing compilation where the correct handling of scoping is required by the function's type. Unfortunately this also means that a great deal of code is spent rearranging the type to be an exact syntactic match with what the compiler expects when it is clearly correct semantically.

As a simple warm-up the compilation of atomic expressions is described first. Recall that this type from figure 5.4 on page 148 consists of numeric literals, variable placement, and addition. As usual variables must be handled using an environment tracking how free variables map to machine values and registers i.e. PHOAS variables.

```
data CompileEnv : Ctx ictx -> MCtx -> Type where
  EmptyEnv : CompileEnv [] mctx
  UExtend : {ctx : Ctx ictx}
         -> (x : Name) -> (t : Ty) -> (i : Ix ictx t)
         -> MVal mctx t
         -> CompileEnv ctx mctx
         -> CompileEnv ((x, (t ** i)) :: ctx) mctx
```

This type is closely related to the environment used for expression evaluation, ensuring that every free variable tracked by an expression's context maps to a well-formed value. However in this case the name, type, and index expression of the variable being mapped are explicit arguments to aid the Idris type-checker. As previously noted only the value's type is respected, avoiding tracking index expressions to simplify the implementation.

Given an environment an atomic expression is compiled and flattened in a single step. Intuitively by compiling an expression, not only is a result produced but additionally a series of new bindings may have been performed. Thus the continuation is not only from a machine value, but additionally from an extension of the context along with the new mappings to values.

```
compileAtomic : {ctx : Ctx ictx}
             -> CompileEnv ctx mctx
             -> AExp ctx t i
             -> ((ctx' : Ctx ictx) ->
                 CompileEnv ctx' mctx ->
                 MVal mctx t ->
                 Asm mctx lctx t')
             -> Asm mctx lctx t'
```

The compilation of literals and variables is trivial, pass the machine value to the continuation without any extension to the context.

```
compileAtomic cenv (NLit n) k = k [] EmptyEnv (Lit (NLit n))
compileAtomic cenv (Var elt) k =
  k [] EmptyEnv (lookupCVar cenv elt)
```

The function `lookupCVar` is the element lookup function which is at this point routine

```
lookupCVar : CompileEnv ctx mctx -> Elem (x, (t ** i)) ctx
          -> MVal mctx t
lookupCVar EmptyEnv pf impossible
lookupCVar (UExtend x t i v cenv) Here  = v
lookupCVar (UExtend x t i v cenv) (There elt) =
  lookupCVar cenv elt
```

Handling addition is only slightly more complex. First either side of the sum are compiled and their results used in an addition instruction whose result is passed to the continuation in the union of the two possibly extended environments.

```
compileAtomic cenv (Add a1 a2) k =
  compileAtomic cenv a1 (\ctx1, cenv1, v1 =>
  compileAtomic cenv a2 (\ctx2, cenv2, v2 =>
  Add v1 v2 (\v3 => k (ctx2 ++ ctx1)
                      (cenv2 `appendCEnv` cenv1)
                      v3)))
```

With the definition of environment union `appendCEnv` given in listing D.1 on page 201. Note that thanks to the strictly defined notion of compilation environment only the shape of the resulting context `ctx2 ++ ctx1` need be known at which point Idris will force the correct environment be constructed.

The definition of the main compile function proceeds similarly, however not only can the expression context be extended but additionally the index expression context can be extended.

```
compile : {ctx : Ctx ictx}
     -> CompileEnv ctx mctx
     -> Stmt facts ctx t i
     -> ((ictx' : IxCtx) ->
         (ctx' : Ctx (ictx' ++ ictx)) ->
         CompileEnv ctx' mctx ->
         MVal mctx t ->
         Asm mctx lctx t')
     -> Asm mctx lctx t'
```

Note that this signature strictly enforces that a context can only be *extended* in `ictx'` and not modified. Intuitively most of the expression forms are relatively simple to compile, however due to embedded scope correctness, a great deal of index manipulation, mostly to manage append associativity, is required. Importantly, as usual, this will make use of thinning and weakening of index expressions in the environment. These functions are a simple case of walking over the collection and applying the respective thinning or weakening operation to each index expression.

```
thinCEnvCtx : CompileEnv ctx mctx -> (ictx' : IxCtx)
           -> CompileEnv (thin ctx ictx') mctx
thinCEnvCtx EmptyEnv ictx' = EmptyEnv
thinCEnvCtx (UExtend x t i v cenv) ictx' =
  UExtend x t (thin i ictx') v (thinCEnvCtx cenv ictx')

weakenCEnvCtx : CompileEnv ctx mctx -> (ictx' : IxCtx)
             -> CompileEnv (weaken ctx ictx') mctx
weakenCEnvCtx EmptyEnv ictx' = EmptyEnv
weakenCEnvCtx (UExtend x t i v cenv) ictx' =
  UExtend x t (weaken i ictx') v (weakenCEnvCtx cenv ictx')
```

Due to the index expression and complexity of the compilation function, only its scheme is provided; however the full proof is included in appendix D.1 on page 201. To start, compilation of atomic expressions is a simple deferment to the function `compileAtomic` in an unextended index expression environment. Handling let is only slightly more difficult, first the statement whose result is to be bound to x is compiled, then the body is compiled in the environment extended with x. Given that compilation is scope safe and type-correct by construction—as was the case in compiling atomics—each continuation will have an associated index expression and variable context, in addition to an environment tracking a type correct mapping of De Bruijn style variables to PHOAS variables. Thus the value resulting from compiling the let body is handled by the continuation in the new context, combining the extensions

from compiling the two statements.

$$\begin{aligned}\mathcal{K}_\varepsilon[\![a]\!]k =& \mathit{compileAtomic}\ a\ (k\ \emptyset)\\ \mathcal{K}_\varepsilon[\![\text{let } x = s_1 \text{ in } s_2]\!]k =& \mathcal{K}_\varepsilon[\![s_1]\!](\lambda\Delta_1, \Gamma_1, \varepsilon_1, v_1.\\ & \mathcal{K}_{(x,v_1),\varepsilon}[\![s_1]\!](\lambda\Delta_2, \Gamma_2, \varepsilon_2, v_2.\\ & k\ (\Delta_2\Delta_1)\ (\Gamma_2\Gamma_1)\ (\varepsilon_2\varepsilon_1)\ (\mathit{Var}\ v_2)))\end{aligned}$$

In the mechanization the contexts Γ_2 and Γ_1 will respectively be well-defined in $\Delta_2\Delta$ and $\Delta_1\Delta$ while the continuation demands a context well-defined in $\Delta_2\Delta_1\Delta$. Noting that Δ is the context associated with the overall let binding, respective applications of thinning by Δ_1 and weakening by Δ_2 are required. Worse still the associativity of these append operations will be incorrect requiring an application of explicit type rewriting. Finally the environments ε_2 and ε_1 being indexed by Γ_2 and Γ_1 respectively suffer from the same problem and thus must be explicitly thinned and weakened, moreover the resulting associativity of operations will be incorrect requiring type coercions. While not a difficult mathematical problem, its rigorous handling introduces a great deal of noise into the compilation process in addition to placing a relatively strong strain on the Idris evaluator.

Handling rewrites is also trivial. Since they have no runtime effect, the underlying statement can be compiled immediately.

$$\mathcal{K}_\varepsilon[\![\text{rewrite } j\ p\ s]\!]k = \mathit{compile}\ s\ k$$

The handling of list operations is relatively simple. In each case a new heap cell is allocated and initialized with *AllocTagged*, returning a pointer. This pointer is the result of the compilation and thus is handed off to the continuation as usual.

$$\mathcal{K}_\varepsilon[\![[]]\!]k = \mathit{AllocTagged}\ \mathit{ConsTyList}\ \mathit{TagListNil}\ \bot\ (\lambda v.k\ \emptyset\ \emptyset\ \emptyset\ (\mathit{Var}\ v))$$

Here the empty list requests a cell tagged by the nil tag and passes the resulting pointer to the continuation in an unextended environment. The use of $\bot$ indicates the fact that nil has no sub-data associated with it and thus no positions can be requested. Compiling a list cons is similar, however

because it has a head and tail as sub-data, these values must be packed into its heap cell at allocation time.

$$\begin{aligned}\mathcal{K}_\varepsilon[\![s_1 :: s_2]\!]k = &\mathcal{K}_\varepsilon[\![s_1]\!](\lambda\Delta_1, \Gamma_1, \varepsilon_1, v_1.\\ &\mathcal{K}_\varepsilon[\![s_2]\!](\lambda\Delta_2, \Gamma_2, \varepsilon_2, v_2.\\ &\mathit{AllocTagged}\ \mathit{ConsTyList}\ \mathit{TagListCons}\ (g\ v_1\ v_2)\ (\lambda v_3.\\ &k\ (\Delta_2\Delta_1)\ (\Gamma_2\Gamma_1)\ (\varepsilon_2\varepsilon_1)\ (\mathit{Var}\ v_3))))\end{aligned}$$

Where g is the cons generating function mapping position requests for the head to v_1 and requests for the tail position to v_2

$$\begin{aligned}g\ v_1\ v_2\ \mathit{Nat}\ \mathit{ConsPosHead} &= v_1\\ g\ v_1\ v_2\ \mathit{List}\ \mathit{ConsPosTail} &= v_2\end{aligned}$$

As usual the resulting list pointer v_3 is defined in an extended scope which is passed into the continuation along with the extended environment.

The compilation of *caselist* is broken down into three cases, one for each possible combination of reachable vs. unreachable branches. Suppose first that both cases are reachable, then compilation may proceed in the usual way. First compile the scrutinee and inspect its tag. In the case the tag denotes an empty list then proceed with compilation in the nil branch. In the non-empty case the values in the list head and tail should be retrieved from the cons pointer, and compilation should proceed in the cons branch.

$$\begin{aligned}&\mathcal{K}_\varepsilon[\![\mathit{caselist}\ s_1\ (x\ xs\ s_2)\ s_3]\!]k = \mathcal{K}_\varepsilon[\![s_1]\!](\lambda\Delta_1, \Gamma_1, \varepsilon_1, (\mathit{Var}\ v_1).\\ &\quad \mathit{GetTag}\ v_1(\lambda t.\\ &\quad \mathit{Jz}\ t\ (\mathcal{K}_\varepsilon[\![s_3]\!](\lambda\Delta_3, \Gamma_3, \varepsilon_3, v_3.\ k\ (\Delta_3\Delta_1)\ (\Gamma_3\Gamma_1)\ (\varepsilon_3\varepsilon_1)\ v_3))\\ &\qquad (\mathit{GetConsPos}\ v_1\ \mathit{ConsPosHead}\ (\lambda v_1^{hd}.\\ &\qquad\ \mathit{GetConsPos}\ v_1\ \mathit{ConsPosHead}\ (\lambda v_1^{tl}.\\ &\qquad\ \mathcal{K}_{(x,v_1^{hd}),(xs,v_1^{tl}),\varepsilon}[\![s_2]\!](\lambda\Delta_2, \Gamma_2, \varepsilon_2, v_2.\\ &\qquad\ k\ (\Delta_2\Delta_1, (x, \mathit{Nat}), (xs, \mathit{List}))\ (\Gamma_2\Gamma_1)\ (\varepsilon_2\varepsilon_1)\ v_2))))))\end{aligned}$$

Note that this is the only the point where an extension of the index expression context Δ occurs, introducing the fresh index expression variables for the head and tail of the unpacked list. As usual the updated contexts and environments are correct with respect to a smaller index expression context and thus must undergo weakening and (sometimes repeated) thinning. As

was the case earlier the resulting associativity of the append operations will be wrong so these too must be corrected by repeated, explicit type coercion. The full proof is in the appendix.

The remaining to cases correspond to either an unreachable cons or nil case, and thus reduce to a compilation of either side of the branch without tag testing. Compilation of each case proceeds just as it did above.

The only remaining statements to compile are the list and nat fold operations. Both being loops which accumulate some result, the compilation is almost identical. First consider the handling of loops on naturals. First the value to loop over must be compiled along with the initial accumulator value. Then each iteration of the loop should first test for its termination condition which in this case is when $n = 0$. Upon termination control is passed to the continuation, otherwise the loop body should be executed, the loop counter decremented, and finally control should jump back to the top of the loop. Recall again that parameterized jumps are used instead of *phi* nodes, thus with each jump to perform another iteration the updated loop values must be passed in.

$$
\begin{aligned}
\mathcal{K}_\varepsilon[\![\mathit{elimnat}\ s_1\ (x, \mathit{acc} \mapsto s_2)\ s_3]\!]k = & \mathcal{K}_\varepsilon[\![s_1]\!](\lambda\Delta_1, \Gamma_1, \varepsilon_1, v_1. \\
& \mathcal{K}_\varepsilon[\![s_3]\!](\lambda\Delta_3, \Gamma_3, \varepsilon_3, v_3. \\
& \mathit{PLabel}\ [v_3, v_1]\ (\lambda l, [v_3', v_1']. \\
& \mathit{Jz}\ (\mathit{Var}\ v_1') \\
& \quad (k\ (\Delta_3\Delta_1)\ (\Gamma_3\Gamma_1)\ (\varepsilon_3\varepsilon_1)\ (\mathit{Var}\ v_3')) \\
& \quad (\mathcal{K}_{(\mathit{acc}, v_3'), (x, v_1'), \varepsilon}[\![s_2]\!](\lambda\Delta_2, \Gamma_2, \varepsilon_2, v_2. \\
& \quad \mathit{Dec}\ (\mathit{Var}\ v_1')\ (\lambda v_1''. \\
& \quad \mathit{PJmp}\ l\ [v_2, v_1'']))))))
\end{aligned}
$$

As earlier remarked, the handling of list iteration is similar to nat iteration. However because the termination criteria is no longer a simple number but a pointer to a structure, the tag is fetched and checked, with 0 being the numeric value of the nil tag. Additionally the value of the head must be bound to a variable and passed into the environment for use in the body. Finally to enter the next iteration of the loop, the tail of the list must be

fetched and passed in.

$$
\begin{aligned}
\mathcal{K}_\varepsilon[\![\mathit{elimlist}\ s_1\ (x, \mathit{acc} \mapsto s_2)\ s_3]\!]k = & \mathcal{K}_\varepsilon[\![s_1]\!](\lambda\Delta_1, \Gamma_1, \varepsilon_1, v_1.\\
& \mathcal{K}_\varepsilon[\![s_3]\!](\lambda\Delta_3, \Gamma_3, \varepsilon_3, v_3.\\
& \mathit{PLabel}\ [v_3, v_1]\ (\lambda l, [v_3', v_1'].\\
& \mathit{GetTag}\ v_1'\ (\lambda t.\\
& \mathit{Jz}\ t\\
& \quad (k\ (\Delta_3\Delta_1)\ (\Gamma_3\Gamma_1)\ (\varepsilon_3\varepsilon_1)\ (\mathit{Var}\ v_3'))\\
& \quad (\mathit{GetConsPos}\ v_1'\ \mathit{ConsPosHead}\ (\lambda v_1^{hd}.\\
& \quad \mathcal{K}_{(acc,v_3'),(x,v_1^{hd}),\varepsilon}[\![s_2]\!](\lambda\Delta_2, \Gamma_2, \varepsilon_2, v_2.\\
& \quad \mathit{GetConsPos}\ v_1'\ \mathit{ConsPosTail}\ (\lambda v_1^{tl}.\\
& \quad \mathit{PJmp}\ l\ [v_2, v_1^{tl}])))))))))
\end{aligned}
$$

Thus by applying the usual strategy of indexing structures by the contexts, the whole compilation scheme is guaranteed to be type and scope-safe. Additionally we note the beauty of Oliviera et al. back-edge encoding, ensuring the type-safety of the parameterized jump call. This not only simplifies the scoping and handling of typed labels, but additionally simplifies the denotational semantics of the abstract machine.

5.3.3 Evaluation and object code emission

The use of PHOAS to encode machine syntax makes evaluation trivial; however due to the encoding of loops, it's not immediately clear how to make evaluation total in the presence of potentially infinite loop. Even were this problem to be solved there's an additional, slightly less obvious issue stemming from *GetConsPos* offering no guarantees about the shape of the value to deconstruct. Thus it's possible to request the head or tail of an empty list! To avoid these problems evaluation is given in the maybe monad using a "timeout" value similar to CompCert. At each step of evaluation a "clock" is consulted: if the clock is zero nothing is returned and evaluation fails, however if the clock is non-zero then evaluation of sub-terms proceeds with a decremented clock. Danielsson [Dan12] gives a similar but finer grained evaluation strategy for the λ-calculus splitting "crashes" and non-termination into the the maybe and partiality monads [Cap05] respectively. The signature of the evaluation function follows.

```
  eval : Asm EvalTy EvalLabel t -> (clk : Nat) -> Maybe (EvalTy t)
```

Since evaluation largely follows the style of the well-typed STLC in section 2.5.4 on page 47 only the handling of labels and lists are considered[1]. In particular the use of PHOAS means that variables are encoded using their underlying semantic data-types during evaluation by `EvalTy`. Thus for example terms constructed by calls to `AllocTagged` are Idris lists, and terms extracted by `GetConsPos` can simply pattern match on the "pointer".

```
eval (AllocTagged ConsTyList TagListCons vf k) (S clk) =
  let v  = evalMVal (vf TyNat ConsPosHead)
      vs = evalMVal (vf TyList ConsPosTail)
  in eval (k (v :: vs)) clk
eval (GetConsPos {t=TyList} (x :: xs) ConsPosHead k) (S clk) =
  eval (k (Var x)) clk
```

The evaluation of labels is another point requiring special handling. By reliance on PHOAS again, labels can be interpreted as functions from their parameter types to values. However as usual this evaluation must be partial since any label can be used to form an infinite loop.

```
  EvalLabel : (n : Nat) -> ParamTypes n -> Ty -> Type
  EvalLabel n ts t = All EvalTy ts -> Maybe (EvalTy t)
```

Thus evaluation of jumps becomes trivial: simply evaluate the packed set of arguments and pass them into the label function. Argument evaluation proceeds by mapping each `MVal` encoded pointer or literal into the underlying semantic value.

```
  evalMLit : MLit t -> EvalTy t
  evalMLit (NLit n) = n

  evalMVal : MVal EvalTy t -> EvalTy t
  evalMVal (Var x) = x
  evalMVal (Lit c) = evalMLit c

  evalArgs : All (MVal EvalTy) ts -> All EvalTy ts
  evalArgs [] = []
  evalArgs (v :: vs) = evalMVal v :: evalArgs vs
```

Then the jump is evaluated by a rather pedestrian call.

```
  eval {t=t} (PJmp {t=t} k args) (S clk) = k (evalArgs args)
```

[1] for the full mechanization see listing D.5 on page 206

This continuation is best appreciated by investigating its construction by the evaluation of a `PLabel`. These "label functions" are constructed by repeatedly unfolding their bodies. Thanks to the use of a continuation based evaluation style the body is free to short-circuit evaluation at any point, thus a label body is blindly unfolded using the current value of the clock.

```
eval (PLabel n ts t args k) (S clk) =
  let vs = evalArgs args
  in bfix (\vs', rec => eval (k rec vs') clk) clk vs
  where bfix : (All EvalTy ts ->
                 (All EvalTy ts -> Maybe (EvalTy t)) ->
                 Maybe (EvalTy t))
            -> Nat
            -> All EvalTy ts
            -> Maybe (EvalTy t)
        bfix f Z     vs = Nothing
        bfix f (S n) vs = f vs (bfix f n)
```

In the above, the body k effectively receives a back-edge *rec* to itself subject to the bounding condition imposed by the clock.

The above means of evaluation corresponds to a more traditional notion of correctness whereby soundness might be specified by showing that for every well-formed source program, evaluation of the compiled code produces a value in `Just` such as

$$\emptyset;\emptyset;\emptyset \vdash s : \tau \wedge p = \mathcal{K}_{\emptyset}[\![s]\!](\lambda\Delta, \Gamma, \varepsilon, v.\ \mathit{Ret}\ v) \implies \exists N\ v. \mathit{eval}\ p\ N = \mathit{Just}\ v$$

Intuitively this should hold since the source program s is known to be terminating. Currently the only areas suffering from potential non-termination are the loops. Since these loops are generated from obviously terminating structural recursion on naturals and lists, the termination proof could perhaps be extracted as an accessibility proof in the style of Bove and Capretta [BC05]. We hypothesize as an area of future work such proofs could be used to internalize a notion of correctness by making the compilation index expression respecting in the style of expression decomposition. This would additionally solve the problem of invalid use of *GetConsPos* since the static shape of an *MVal* value would always be known.

Given the above formulation PHOAS is repeatedly leveraged to simplify the compilation process, directly ensuring the safe usage of scoped values without the need of an explicitly typed side environment. PHOAS additionally allows for the natural encoding of loops using a bounded fixpoint operator `bfix`. As a final display of the flexibility granted by PHOAS

and the completeness of the compilation stack, a brief description of code emission as "low-level" C follows.

Emitting object code

For simplicity a final translation from abstract machine to object code is performed via C. LLVM should also be readily targetable, however this would require a further translation from parameterized jumps to ϕ instructions whereas such jumps can be efficiently encoded in C directly using the preprocessor.

Formatting and rendering the C output follows the style of Hughes [Hug95] and Wadler [Wad98], defining the `Doc` and `SimpleDoc` types decorating text with a basic layout structure allowing for nesting and column restricted rendering.

While the translation is largely trivial e.g. formatting an `Add` instruction as C's "+" operator applied to its formatted values, rendering jumps is unobvious. Labels can be treated exactly like functions, however since calls to labels are guaranteed to occur in tail positions arguments can be passed by overwriting its own frame. Thus by further appeal to PHOAS both variables and labels should be interpreted as names or strings.

```
RName : Ty -> Type
RName = const Name

LName : LabelCtx
LName n ts t' = Name
```

Allowing for the typing of a core rendering function.

```
renderC : Asm RName LName t -> NameSupply Doc
```

Here the use of `NameSupply` as explored by Augustsson [ARS94] is required to generate unique names in e.g. `Jz` where the number of names required by the left and right branches are not known. The monad provides a function `fresh` for generating new, unique names.

As implied, labels are handled by generating a "frame", i.e. the label's parameters, and a C label accessible via the usual `goto` statements. Upon entry the label's frame variables are initialized from the values placed in the `PLabel`. Then when a jump is performed, the arguments to the label overwrite the currently active frame values and the jump is performed with `goto` ensuring efficient evaluation in both time and space. Thus rendering a jump is effectively rendering a C call

```
renderPLabelSetup : Name -> Args RName ts -> Doc
renderPLabelSetup l vs = text (plabelSetupName l)
                       <> wrap "(" (renderArgs vs) ")"

renderC (PJmp l args) =
  pure (renderPLabelSetup l args `renderSeq`
        text ("goto " ++ l ++ ";"))
```

Then the rendering of a labeled block is coarsely broken into three steps

1. Generate fresh names for the frame variables and declare the label's parameters using these names.

2. Generate a new macro installing its arguments into the frame variables associated with the label.

3. Initialize the label's parameters using the values packed into the `PLabel` and render the label's body under a fresh C label.

```
renderC (PLabel n ts t args k) = do
  l <- fresh
  let frameVars = genFrameVars l Z n ts
  prog <- renderC (k l frameVars)
  pure (renderDecls frameVars args `renderSeq`
        renderPLabelSetupDecl l n <> line <>
        (renderPLabelSetup l args `renderSeq`
         text (l ++ ":") <+> bracket "{" prog "}"))
```

Thus the resulting code in a two argument label with initial values 0 and 1 would be

```
#define plabel_setup_v0(l0, l1) \
  v0Frame1 = l1; v0Frame0 = l0;
plabel_setup_v0(0, 1);
v0: {
  ...
  plabel_setup_v0(v3, v2);
  goto v0;
}
```

As with the handling of `Add`, the remainder of the cases are straightforward with the handling of list allocation performed directly in C via a simple run-time system (RTS). To help gather intuition about this compiler output it's useful to examine several simpler cases. Extending the assembly language with an explicit return instruction

```
data Asm : MCtx -> LabelCtx -> Ty -> Type where
  ...
  Ret : MVal mctx t -> Asm mctx lctx t
```

makes it easy to inspect the output of the compiler. The additional convenience function is also defined, allowing for the easy dumping of compiled (closed) program strings.

```
dump : Stmt facts [] t i -> String
dump s = prettyC (compile EmptyEnv s (\ictx', ctx', cenv', v =>
                                        Ret v))
```

In a simple instance, let bindings will produce a C variable and addition will produce C addition, e.g. the program adding 40 to $x = 2$ below

```
Let "x"  (Atomic (NLit 2)) (Atomic (NLit 40 `Add` Var  "x"))
```

produces

```
int v0 = 40 + 2;
return v0;
```

Considering a slightly more complicated program which binds multiple variables maps to a C program with multiple binding, e.g.

```
Let "x" (Atomic (NLit 2))
    (Let "y" (Atomic (NLit 40 `Add` Var  "x"))
         (Atomic (NLit 20 `Add` Var "y")))
```

produces

```
int v0 = 40 + 2;
int v1 = 20 + v0;
return v1;
```

Although programs operating on numbers are relatively simple, even the most trivial program working with lists poses a considerable increase in complexity. This is because all list operations must call into the RTS to allocate cells, and mark them with the proper type of constructor e.g. to allow for case analysis of a list, or for garbage collection. To begin, consider the program which returns the two element list `3 :: 2 :: []`

```
Let "x"  (Atomic (NLit 3) :: Atomic (NLit 2) :: [])
         (Atomic (Var "x"))
```

First the system will allocate the empty list, a constructor with zero positions.

```
cons_t v0 = new_cons( 0);
set_cons_tag( 1);
;
```

Note that the nil constructor is marked with the unique tag: 1, chosen by the compiler. This allows the RTS to inspect the element in the heap and discover that it represents the empty list constructor.

Next the system will allocate a cons cell, brand it with the tag unique to the list cons constructor (i.e. 0), and then set its two positions: i.e. the head and tail to 2 and the previously constructed nil list respectively.

```
cons_t v1 = new_cons( 2);
set_cons_tag( 0);
set_cons_pos( v1, 0, 2);
set_cons_pos( v1, 1, v0);
;
```

As expected, position 0—the head—is set to the constant numeric value 2, while position 1—the tail—is set to the nil list which was previously bound to *v0*.

Finally the C program will allocate the left-most cons cell `3 :: ...`, setting it's head and tail to the constant 3, and the cons cell allocated above.

```
cons_t v2 = new_cons( 2);
set_cons_tag( 0);
set_cons_pos( v2, 0, 3);
set_cons_pos( v2, 1, v1);
;
return v2;
```

The initialization of the tag and positions follows the pattern of the first cons cell, and finally the program returns the constructed list in the temporary *v2*.

Having the ability to easily dump C programs also allows us to see how the compiler uses reachability informantion to optimize its output. For example compiling the safe tail function which only accepts a non-empty list won't produce a branch. Suppose safe head is called on the following above list, i.e.

```
Atomic (NLit 3) :: Atomic (NLit 2) :: []
```

Then the output C will first allocate a nil list, followed by two cons cells as was the case above

```
cons_t v0 = new_cons(0);
set_cons_tag( 1);
;
```

```
cons_t v1 = new_cons( 2);
set_cons_tag( 0);
set_cons_pos( v1, 0, 2);
set_cons_pos( v1, 1, v0);
;
cons_t v2 = new_cons( 2);
set_cons_tag( 0);
set_cons_pos( v2, 0, 3);
set_cons_pos( v2, 1, v1);
;
```

However rather than generating an *if* statement checking whether the input list is nil or not, instead the compiler will simply compile the reachable branch, grabbing the head and tail, and producing the head

```
int v3 = get_cons_pos( v2, 0);
cons_t v4 = get_cons_pos( v2, 1);
return v3;
```

The first call to *get_cons_pos* grabs positions 0—i.e. the list head—of the list pointed to by temporary *v2* and stores the result in *v3*. Finally the compiler grabs the tail from position 1 before returning the list head, *v3*, as requested.

As a final example, it's useful to see concretely how the tag information is used by the compiler e.g. by performing case inspection of a list. Consider the unsafe head function, i.e. the head function which accepts both a non-empty list, *or* an empty list, in which case a default value of 0 is returned.

```
CaseList xs
  "x" "xs'" (Reachable
               (CanReach
                  [RequireEq (var "i") (var "x" :: var "xs'")]
                  (\[hd,tl,i], _, [Refl] => Refl))
               (var "x"))
  (Reachable
     (CanReach
        [RequireEq (var "i") []]
        (\ixEnv, _, [iEnil] => iEnil))
     Z)
```

Applying this function to the usual list `3 :: 2 :: []` will result in C which first constructs the list, and then immediately inspects the tag to deconstruct it. First the list construction

```
cons_t v0 = new_cons( 0);
set_cons_tag( 1);
;
cons_t v1 = new_cons( 2);
set_cons_tag( 0);
set_cons_pos( v1, 0, 2);
set_cons_pos( v1, 1, v0);
;
cons_t v2 = new_cons( 2);
set_cons_tag( 0);
set_cons_pos( v2, 0, 3);
set_cons_pos( v2, 1, v1);
;
```

Next the C program will grab the constructor tag from the list stored in *v2* in order to decide if it is nil—i.e. the tag is 1—or non-nil—i.e. the tag is 0.

```
int v3 = get_cons_tag( v2);
if (v3) {
  int v4 = 0;
  return v4;
} else {
  int v5 = get_cons_pos( v2, 0);
  cons_t v6 = get_cons_pos( v2, 1);
  return v5;
}
```

As expected, in the nil case the compiler returns a value of 0 bound to the temporary *v4*; and in the non-nil case the compiler first projects the head and tail into *v5* and *v6* respectively before finally returning the head in *v5*.

5.4 Summary

In this chapter an example use case of the index expressions developed in chapter 3 was given, presenting a means of statically costing expressions. The analysis presented a method of using the denotation of index expressions ‘—given in listing 3.2 on page 80—to perform a symbolic evaluation required e.g. in the calculation of loop and case costs. Moreover to account for simple optimizations such as machine support for natural numbers, the costing is

performed over an intermediate statement language $\mathscr{S}$ rather than directly on $\mathscr{L}$.

To perform compilation to reasonably efficient machine code, a verified decomposition function is given compiling from $\mathscr{L}$ to $\mathscr{S}$. It was shown how the correctness would follow indirectly as a consequence of the decomposition function being index expression respecting, and thus from the canonical values property a index expression respecting evaluation entails that the transformation is semantics preserving.

Chapter 6

Conclusions

In this thesis we have explored the design and implementation of a simple language as within a dependently typed context. We have shown that languages without syntax directed typing rules can be elegantly dealt with courtesy of the affordances offered by dependent types and their capacity to *interactively* perform work, deferring to the programmer when necessary. Two concepts have been revisited throughout this work focusing on how the use of constraints and reflection can be used to aid programming in a correct-by-construction style; and how their use with synthetic dependent types simplify the process of reasoning about programs. Throughout the course of developing $\mathscr{L}$ and exploring its use key contributions were made using powerful features exposed by the host.

Dependent types are a convenient host for not only the embedding of a language and compiler, but additionally of sophisticated type systems.

Index expressions are used to statically capture the semantics of a term allowing for reasoning to be performed such as the determination of a case branches reachability. Such proofs make use of a novel constraint reflection mechanism. Constraints are built up by the iterative refinement of input values using case analysis.

Explicit coercions provide a means of using the host's reduction semantics to perform checking of index expressions. The coercions use locally available evidence from the constraints making them correct by construction.

Sound compilation shows the feasibility of the structure of $\mathscr{L}$, making use of the index expressions to ensure the correctness of AST transformations. In particular the full compilation stack performs several simple, optimizations which are shown to be sound. First high-level operation on natural numbers are extracted; then peano numbers are eliminated allowing for their compilation to machine integers. After optimizing the representation of numbers the statement language $\mathscr{S}$ is compiled to a simple abstract machine which can be executed directly or used to produce architecture native object code.

6.1 Limitations

While the initial development of $\mathscr{L}$ supports a variety of interesting features, there remain a few key areas to explore.

Side-effects and in particular programs with I/O are not considered in this thesis.

Performance improvements particularly relating to type-checking are necessary to scale the the current implementation of $\mathscr{L}$ where a powerful machine is required for relatively small programs.

Data structures are explored in a limited capacity and user-defined types are not available in $\mathscr{L}$.

6.1.1 Supporting side-effects

While the language presented in this thesis doesn't support I/O we think that none of the ideas presented are fundamentally incompatible with the more general concept of side-effects. For instance the algebraic effects [Bra14b]; [BP12] style of introducing specialized syntax allows for opaque function calls; however calls could additionally introduce arbitrary constraints into the context for use by the programmer. Since the interpreter must prove that any proposed constraints are satisfiable there is no fear of an ill-behaved side-effect. Such a system could further be leveraged to provide a GHC style plugin architecture where 3$^{\text{rd}}$ party checkers can always be trusted; currently only limited support is available.

The strategy of cost analysis should also scale to support side-effects. Motivating this is the opaque cost applied to "atomic" operations. We believe opaque system calls could also be treated atomically.

Currently the low-performance of the $\mathscr{L}$ implementation places practical limitations on the scope of further language extensions.

6.1.2 Improving performance

One of the primary problems with the language presented in this thesis is its lack of scalability due to performance. For example verifying the cost of a loop encoded sum takes several minutes and gigabytes of ram on a boosted 4GHz Haswell machine. We believe the performance problems in $\mathscr{L}$ rest primarily in the evaluation performance of the host and in the choice of variable encoding in $\mathscr{L}$ itself.

Host evaluation performance

The reason for Idris' comparatively low evaluation performance is because development efforts have so far emphasized features that allow for the exploration of developing larger systems with dependent types. By comparison Coq's calculus of inductive construction [CP90] supports dependent types but is primarily meant for real world verification of software [Gon07]; [KLW14]; [Gon+13]. Therefore comparatively little effort has been put into supporting programming paradigms making heavy use of dependent types [Soz10] which are still considered niche [Ler09]. In particular during the development of the proof of the four-color theorem [Gon07], Coq too ran into evaluation related performance problems and was modified to use a compiled implementation of strong reduction [GL02]. This was an important performance optimization for large scale proofs by reflection like the four-color theorem which invokes a decision procedure for determining the 4-colorability of a graph on every possible input [GL02]. The implementation is flexible in that it allows for checking β-equality of terms while still based on compilation targeting a highly efficient byte-code interpreter [Ler95]. This implementation of term reduction is up to 100 times faster than both Coq's call-by-name and call-by-value interpreters. More recently an untyped normalization by evaluation strategy for efficiently testing term convertibility in dependent type theories was developed by Boespflug [Boe10] using Haskell.

Such an optimization could certainly be adapted to the implementation of Idris which would alleviate many of the performance concerns.

Although such an approach would be beneficial to the developments in this thesis and the wider Idris ecosystem, it's also a potentially large undertaking. Another means of improving performance to be done alongside Idris enhancements is to find points needing optimization in the encoding of $\mathscr{L}$ itself.

Naïve syntax encoding

While the choice of variable encoding in $\mathscr{L}$ is convenient from a reasoning perspective since manipulations are clarified in the type, it also requires values well-formed in a context undergo explicit thinning upon any extensions. Although Idris does not currently support profiling of evaluation, the current implementation of $\mathscr{L}$ makes use of widespread thinning, triggering repeated data structure traversals throughout evaluation which are believed to cause a performance bottleneck.

One possible strategy to consider is for improving performance is the weakening of entire trees in the style of Bird and Patterson [BP99]. This strategy should be directly available without the need for an encoding via a polymorphic type. Instead the binding context at the type-level can be directly referenced and extended.

Another possible strategy is to use PHOAS, avoiding the need for any form of context manipulation since unbound names are trivially inaccessible.

Preliminary work on improving performance within a simplified calculus has yielded unscientific improvements of around $1,000$ times in critical areas.

A drastic improvement in performance should allow for the exploration of further features such as user-defined data types and parallel matching.

6.1.3 Supporting data structures

The current implementation of $\mathscr{L}$ has builtin supports for lists and natural numbers and does not allow for user defined data-types. The primary hurdle in supporting user-defined types is in the derivation of constraints generated during case analysis where e.g. those generated by list destruction are hard-coded. The use of a universe of data-types such as the polynomial functors capturing SML style sums-of-products, or the container types would provide a wide range of definable structures. Moreover because the data-structure

"codes" are first order terms, constraints could be programmatically derived. This would additionally simplify the syntax of the language by unifying the case and fold statements at the expense of a more complicated type, requiring their introduced constraints be calculated. While this is likely feasible the further burden on evaluation would first require an exploration of performance enhancements discussed above.

6.2 Further work

The use of dependent types trivially allows for the type-checking of systems without syntax directed rules, or even an obvious global analysis. The use of explicitly tracked constraints as in the types of $\mathscr{L}$ terms to aid in this process allows for the separation of program implementation and proof development in a convenient way. This allows for skilled developers without experience in formalized theorem proving to make assumptions and naturally defer the proof of validity to a machine via a tactic or another member of their team. Two primary areas of further investigation immediately emerge

1. An exploration of expanded programming features.
2. Developing a correct-by-construction proof of compilation correctness.

As discussed above the current implementation has hard-coded type support limited to lists and integers. A natural extensions of this idea is to introduce a more expressive universe such as the polynomial functors or container types.

6.2.1 Expansion of programming features

The language $\mathscr{L}$ proves the feasibility of interactively proving properties about programs using semantic indexing in the context of dependent types. However the current implementation has hard-coded type support limited to lists and integers. As discussed in the limitations, expanding the scope to a general class of types is an obvious target of exploration.

Extensions to new type systems

Although the current usage of constraint reflection for proving the type correctness of expressions is restricted to working with simple types, there

is no fundamental reason this should be the case. The inclusion of lists in $\mathscr{L}$ exemplifies the flexibility of the approach in handling structured data. Thus an interesting area of work would be the application to the variant of System F_ω described by Morrisset [Mor06] capturing SML modules and Haskell typeclasses.

Closely related is the system of LXres. In particular LXres like F_ω uses its sophisticated kind system to calculate types, restricting the possible shape of runtime values. Crary and Weirich note that this places a requirement on the programmer to carefully represent types in a way which is "not too abstract". In contrast using a constraint based approach would allow for a more direct encoding of these restrictions at the expense of decidable type-checking.

This use of constraints naturally begs the question of its application in formalizing the correctness of constraint based systems such as F^{RGN} and λ^{rgnUL} of Fluet et al. [FMA06] and the closely related language Rust. These systems allow safe, low-level access to memory by the use of region and lifetime systems. Regions represent a slice of memory, whereas lifetimes represent the scope of a region's use. Moreover these types can be coerced e.g. if a region is expanded then a notion of subtyping applies, and these two regions likely have related lifetimes. The approach of using constraints to prove the validity of coercions in $\mathscr{L}$ should adapt to support correct by construction notions of subtyping.

While all of the above are relatively expressive languages, it would be interesting to apply the strategies developed in this thesis to an EDSL such as Feldspar. In particular targeting areas such as hard real-time systems which often require strong guarantees but use relatively simple programming constructs might introduce the possibility of more automated rewrite tactics. As an example in $\mathscr{L}$ because index expressions themselves are first order, an *index expression reflection* procedure could be developed without the need for explicit first-class reflection support. Index expressions can be automatically reflected into an underlying monoid expression consumable by the monoid solver. Hard real-time systems could also provide an opportunity to expand the scope of the costing framework developed in chapter 5 on page 136.

Probabilistic cost analysis

One of the interesting aspects of the cost analysis developed in chapter 5 is the use of an opaque costing function at statement leaves. Such a system is amenable to a probabilistic analysis, loosely coupled with the underlying ISA. Rather than assigning an identifiable, discrete cost to atomic expressions, instead they could be drawn from some density function. While this abandons any certainty that might be required by a soft or hard real-time system, surprisingly the introduced error can be bounded using well understood statistical techniques. If the tail of the cost's density function is assumed to decay rapidly enough (e.g. sub-gaussian) then estimates of the underlying value can be made to deviates by at most ϵ with confidence δ. This fact follows from an application of Hoeffding's inequality and could be included directly in the cost trace. For instance a trace could provide some likely cost along with a proof that this value deviates from some "true cost" by ϵ with likelihood no more than δ. This is possible by treating accuracy and confidence like a resource. At the top-level some finite amount of accuracy and confidence are available which then are split between sub-trees and sprinkled into the leaves as required. Such a cost guarantee is given with respect to the cost of executing on the machine which gives the "true cost". In our case this could be given using the abstract machine defined in section 5.3 on page 158. If the compilation scheme were shown to be sound the system could provide a strong guarantee of correctness in both a program's functional and non-functional aspects.

Given these additional proofs of correctness, the only remaining piece of the compilation stack which is not known to be sound is the final compilation process from statements to abstract machine.

6.2.2 Proof of sound compilation process

While the current compilation process gives a strong guarantee of safety with respect to type-preservation it lacks two further important properties: correct-by-construction index expression erasure, and proof of soundness. Solving both of these problems should be tractable.

In order to prove the safety of index expression erasure, rather than compiling directly from $\mathscr{S}$ a new intermediate form should be introduced without support for explicit coercions. This should not be a problem since currently the only occurrence of rewrites in a statement resulting from a

decomposed expression are those introduced by case expressions. In fact, rewrites are nearly erased by the current decomposition scheme since all existing rewrites are composed and pushed to a single, top-level rewrite. The primary issue identified in section 5.2 on page 146 is the failure to eliminate the translated index expressions of decomposed expressions from the retained constraint context.

Intuitively the issue of stale index expressions in the constraint context should be solvable because at each step of rewriting only a single index expression is transformed. Thus rather than producing a statement well-defined in the input constraint context Ξ, some new constraint context Ξ' is produced. The soundness of such an approach should follow in the same style as the rewritten index expression j. Since only one index expression is ever transformed, a stack of proofs transporting between identified index expressions could be used to identify constraint contexts. Intuitively this stack encodes the sequence of rewrites necessary to perform at each level of the input constraint context to arrive at the output context. This would also give us a rewrite minimality law stating that

Hypothesis 6.2.0.1 (Rewrite minimality)**.** *Every expression of index expression i with rewrites in Ξ can be decomposed into an expression of index expression j without rewrites such that $\exists \Xi'.i \approx^{\Xi'} j \wedge \Xi \approx \Xi'$*

In fact, the author has made great progress in mechanizing this proof; however due to efficiency problems in the current implementation, attempting to complete the proof results in an explosion of memory consumption.

Given a notion of terms without rewrites the validity of their erasure is obvious without the need for an auxiliary proof that evaluation agrees on terms with and without index expressions. Compilation from a language without rewrites would then only need to be index expression respecting in order to be constructively sound. That is to say internalizing soundness requires the semantic index of the source be maintained. Since rewrites are no longer available, the constraint context no longer needs to be tracked and thus the only non-trivial cases are the looping constructs: list and nat iteration. As discussed in section 5.3.2 on page 163 a possible solution is to generate accessibility proofs from the obviously terminating folds. This fixes not only the termination issue in the evaluation function but additionally the issue stemming from the potential mis-application of *GetConsPos* since all values have static access to their shape.

The developments in this thesis using well-understood dependently typed paradigms have worked towards a sound compilation stack for a simple language. Despite great strides in theorem proving technology, mechanizing correctness and experimenting with type-systems as in the POPLmark challenge [Ayd+05] still requires expertise in using theorem provers. We think the use of constraints and explicit coercions form an interesting basis for mechanizing type-systems with non-trivial reduction rules, allowing skilled programmers to work as usual and deferring necessary proofs to custom tailored tactics or other members of the team. Still, more work is required to explore a broader range of constraints—such as linear inequalities—and to sufficiently automate common rewrites such as the usual style of reduction in dependently typed languages. We would like to eventually work towards a system which allows for a mixture of completely automated rewrites by computing over the constraints in context, and towards strategies which defer difficult to solve proofs e.g. as is done by Coq's `Program Definition`. Such a system could produce a term which proves some proposition given proof of several constraints it describes as pure data `(cs : Constraints ctx) -> (EvalConstraints cs -> Dec p)`.

Appendix A

Auxiliary correctness proofs of $\mathscr{L}$ programs

A.1 Correctness of `ixSnd`

```
ixSndCorrect : (xs : List Nat) -> ixSndEval xs = listSnd xs
ixSndCorrect (x :: y :: _) = Refl
ixSndCorrect (x :: _) = Refl
ixSndCorrect [] = Refl
```

A.2 Implementation of `expListSnd`

```
expListSnd {facts=facts} {i=i} {hd1=hd1} {tl1=tl1}
           {hd2=hd2} {tl2=tl2} e =
  CaseList e
    hd1 tl1 (Reachable
                (CanReach
                   [RequireEq (var i) (var hd1 :: var tl1)]
                   (\(hdv :: tlv :: ixEnv), _, [iEhdtl] => iEhdtl))
                (CaseList (var tl1)
                   hd2 tl2
                   (Reachable
                     (CanReach
                       [RequireEq (var tl1) (var hd2 :: var tl2)]
                       (\ixEnv, _, [tl1Ehd2tl2] => tl1Ehd2tl2))
                     (var hd2))
                   (Reachable
                     (CanReach
                       [RequireEq (var tl1) []]
```

```
                    (\ixEnv, _, [tl1Enil] => tl1Enil))
                  Z)))
    (Reachable
      (CanReach
         [RequireEq (var i) []]
         (\ixEnv, _, [iEnil] => iEnil))
      Z)
```

A.3 Specification of fibonacci function

```
fibEval : Nat -> List Nat
fibEval =
  natFold (\m, vacc =>
             [listSndEval vacc
             ,listFstEval vacc `plusFold` listSndEval vacc])
          [Z, S Z]

fibEvalStepCorrect
  : (n : Nat) -> (acc : List Nat)
 -> (\m, vacc =>
      [listSndEval vacc
      ,listFstEval vacc `plusFold` listSndEval vacc]) n acc
      =
      fibFoldStep n acc
fibEvalStepCorrect n acc =
  replace {P=\v =>
      [listSndEval acc
      ,listFstEval acc `plusFold` listSndEval acc] =
      [listSnd acc, v]}
    (plusFoldSpec (listFst acc) (listSnd acc))
    (replace {P=\v =>
          [listSndEval acc
          ,listFstEval acc `plusFold` listSndEval acc] =
          [v, listFst acc `plusFold` v]}
        (listSndDestruct acc)
        (replace {P=\v =>
             [listSndEval acc
             ,listFstEval acc `plusFold` listSndEval acc] =
             [listSndEval acc
             ,v `plusFold` listSndEval acc]}
           (listFstDestruct acc)
           (Refl {x=[listSndEval acc
                    ,listFstEval acc `plusFold`
                     listSndEval acc]})))
```

```
fibEvalCorrect : (n : Nat) -> fibEval n = fibFold n
fibEvalCorrect n =
  natFoldFSubst
    {f = \m, vacc =>
      [listSndEval vacc
      ,listFstEval vacc `plusFold` listSndEval vacc]}
    {g = fibFoldStep}
    fibEvalStepCorrect n [Z, S Z]

fibSpec : (i : Ix ictx TyNat)
       -> eval (fibIx i) ixEnv = fibFold (eval i ixEnv)
fibSpec {ixEnv=ixEnv} i = fibEvalCorrect (eval i ixEnv)
```

Appendix B

Implementing a monoid of addition

B.1 Correctness of `plusEval`

```
plusEvalCorrect : (m, n : Nat) -> plusEval m n = m + n
plusEvalCorrect Z     n = Refl
plusEvalCorrect (S m) n = cong (plusEvalCorrect m n)
```

Listing B.1: Monoid laws for addition

```
plusEvalZeroRightNeutral : (m : Nat) -> m `plusFold` Z = m
plusEvalZeroRightNeutral m =
  replace {P=\v => v = m}
          (sym (plusEvalCorrect m Z))
          (plusZeroRightNeutral m)

plusEvalFoldAssoc : (m1, m2, m3 : Nat)
                 -> plusFold m1 (plusFold m2 m3) =
                    plusFold (plusFold m1 m2) m3
plusEvalFoldAssoc m1 m2 m3 =
  replace {P=\v => plusFold m1 (plusFold m2 m3) = v}
    (sym (plusEvalCorrect (plusFold m1 m2) m3))
    (replace {P=\v => v = plusFold m1 m2 + m3}
       (sym (plusEvalCorrect m1 (plusFold m2 m3)))
       (replace {P=\v => m1 + v = plusFold m1 m2 + m3}
          (sym (plusEvalCorrect m2 m3))
          (replace {P=\v => m1 + (m2 + m3) = v + m3}
             (sym (plusEvalCorrect m1 m2))
             (plusAssociative m1 m2 m3))))
```

B.2 Proof of `eqMonoidExp`

```
eqMonoidExp' : (x, y : a)
            -> (x' : a ** (NfMonoidExp m ctx x', x = x'))
            -> (y' : a ** (NfMonoidExp m ctx y', y = y'))
            -> Maybe (x = y)
eqMonoidExp' x y (x' ** (x'Nf, xEx')) (y' ** (y'Nf, yEy')) =
  case eqNfMonoidExp x'Nf y'Nf of
    Just x'Ey' =>
      Just (replace {P=\v => v = y} (sym xEx')
                    (replace {P=\v => x' = v} (sym yEy') x'Ey'))
    Nothing => Nothing

eqMonoidExp : MonoidExp {a=a} m ctx x
           -> MonoidExp m ctx y
           -> Maybe (x = y)
eqMonoidExp e1 e2 =
  eqMonoidExp' _ _ (normMonoidExp e1) (normMonoidExp e2)
```

B.3 Definition of the explicit monoid interface

```
interface ExplicitMonoid (a : Type) where
  total unit : a
  total append : a -> a -> a
  total appendAssociative : (x, y, z : a)
                         -> append x (append y z) =
                            append (append x y) z
  total unitLeft : (x : a) -> append unit x = x
  total unitRight : (x : a) -> append x unit = x
```

Appendix C

Mechanization of index expression evaluation and correctness

C.1 Ix evaluation

```
listElim : (p : List a -> Type)
        -> ((x : a) -> (xs' : List a) -> p xs' -> p (x :: xs'))
        -> p []
        -> (xs : List a)
        -> p xs
listElim p pIH pNil [] = pNil
listElim p pIH pNil (x :: xs') =
  pIH x xs' (listElim p pIH pNil xs')

listDestruct : {a : Type}
            -> (p : List a -> Type)
            -> (kCons : (y : a) -> (ys : List a) -> p (y :: ys))
            -> (kNil : p [])
            -> (xs : List a)
            -> p xs
listDestruct p kCons kNil xs =
  listElim p (\x, xs', pxs' => kCons x xs') kNil xs

listFold : (a -> b -> b) -> b -> (xs : List a) -> b
listFold {b=b} f b0 xs =
  listElim (const b) (\x,xs',acc => f x acc) b0 xs
```

```
natElim : (p : Nat -> Type)
       -> ((n : Nat) -> p n -> p (S n))
       -> p Z
       -> (n : Nat)
       -> p n
natElim p pIH p0 Z     = p0
natElim p pIH p0 (S n) = pIH n (natElim p pIH p0 n)

natFold : ((n : Nat) -> a -> a) -> a -> Nat -> a
natFold {a=a} f x n = natElim (const a) f x n
```

C.2 Expressions with constraints

$\boxed{\Delta; \Xi; \Gamma \vdash e : \tau\ S}$ (well-formed, indexed expression with constraints)

$$\frac{\Delta; \Gamma \ni_v (x, \tau, i) \quad \Delta \vdash i : \tau}{\Delta; \Xi; \Gamma \vdash v : \tau\ i}$$

$$\frac{}{\Delta; \Xi; \Gamma \vdash 0 : \mathit{Nat}\ 0} \qquad \frac{\Delta; \Xi; \Gamma \vdash e : \mathit{Nat}\ i}{\Delta; \Xi; \Gamma \vdash e + 1 : \mathit{Nat}\ (i + 1)}$$

$$\frac{}{\Delta; \Xi\Gamma \vdash [] : \mathit{List}\ []} \qquad \frac{\Delta; \Xi; \Gamma \vdash e_1 : \mathit{Nat}\ i_1 \quad \Delta; \Xi; \Gamma \vdash e_2 : \mathit{List}\ i_2}{\Delta; \Xi; \Gamma \vdash e_1 :: e_2 : \mathit{List}\ (i_1 :: i_2)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathit{List}\ i_1 \\ (j, \mathit{Nat}), (\mathit{js}, \mathit{List}), \Delta; (i_1 \sim j :: \mathit{js}), \Xi; (x, \mathit{Nat}, j), (\mathit{xs}, \mathit{List}, \mathit{js}), \Gamma \vdash e_2 : \tau\ i_2 \\ \Delta; (i_1 \sim []), \Xi; \Gamma \vdash e_3 : \tau\ i_3\end{array}}{\Delta; \Xi; \Gamma \vdash \text{caselist}\ e_1\ (x :: \mathit{xs}) \Rightarrow e_2;\ [] \Rightarrow e_3 : \tau\ (\mathit{caselist}\ i_1\ (j :: \mathit{js}) \Rightarrow i_2;\ [] \Rightarrow i_3)}$$

$$\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash e_1 : \mathit{List}\ i_1 \\ (j, \mathit{Nat}), (\mathit{acc}, \tau), \Delta; \Xi; (x, \mathit{Nat}, j), (\mathit{xs}, \tau, \mathit{acc}), \Gamma \vdash e_2 : \tau\ i_2 \\ \Delta; \Xi; \Gamma \vdash e_3 : \tau\ i_3\end{array}}{\Delta; \Xi; \Gamma \vdash \text{elimlist}\ e_1\ (x, \mathit{xs} \mapsto e_2)\ e_3 : \tau\ (\mathit{elimlist}\ i_1\ (j, \mathit{acc}) \mapsto i_2)\ i_3)}$$

$$\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash e_1 : \mathit{Nat}\ i_1 \\ (j, \mathit{Nat}), (\mathit{acc}, \tau), \Delta; \Xi; (x, \mathit{Nat}, j), (\mathit{xs}, \tau, \mathit{acc}), \Gamma \vdash e_2 : \tau\ i_2 \\ \Delta; \Xi; \Gamma \vdash e_3 : \tau\ i_3\end{array}}{\Delta; \Xi; \Gamma \vdash \text{elimnat}\ e_1\ (x, \mathit{xs} \mapsto e_2)\ e_3 : \tau\ (\mathit{elimnat}\ i_1\ (j, \mathit{acc}) \mapsto i_2)\ i_3)}$$

C.3 Proof of index erasure isomorphism

```
leftInvNat : (n : Nat)
          -> erase {t=TyNat} (reconstructNat n) = n
leftInvNat Z = Refl
leftInvNat (S n) = cong (leftInvNat n)

rightInvNat : (n : IxNat i)
           -> reconstructNat (erase {t=TyNat} n) = n
rightInvNat Z = Refl
rightInvNat (S n) = cong (rightInvNat n)

leftInvList : (xs : List Nat)
           -> erase {t=TyList} (reconstructList xs) = xs
leftInvList [] = Refl
leftInvList (x :: xs) = congCons (leftInvNat x) (leftInvList xs)

rightInvList : (js : IxList xs)
            -> reconstructList (erase {t=TyList} js) = js
rightInvList [] = Refl
rightInvList {xs=x::xs}(j :: js) with (rightInvNat j)
  rightInvList {xs=x::xs} (reconstructNat x :: js) | Refl
    with (rightInvList js)
    rightInvList {xs=x::xs} (reconstructNat x :: reconstructList xs)
      | Refl | Refl = Refl
```

C.4 Proof of indexed value canonicity

```
canonicalNat : (i, i' : IxNat m) -> i = i'
canonicalNat {m=Z}   Z     Z      = Refl
canonicalNat {m=S m} (S i) (S i') =
  cong (canonicalNat {m=m} i i')

consCong : {j, j' : IxNat m}
        -> {js, js' : IxList xs}
        -> j = j'
        -> js = js'
        -> j :: js = j' :: js'
consCong Refl Refl = Refl

canonicalList : (i, i' : IxList xs) -> i = i'
canonicalList {xs=[]} [] [] = Refl
canonicalList {xs=x :: xs} (j :: js) (j' :: js') =
  consCong (canonicalNat j j') (canonicalList js js')
```

```
EvalIxTy : (t : Ty) -> EvalTy t -> Type
EvalIxTy TyNat  = IxNat
EvalIxTy TyList = IxList

canonicalValue : (v, v' : EvalIxTy t i) -> v = v'
canonicalValue {t=TyList} v v' = canonicalList v v'
canonicalValue {t=TyNat}  v v' = canonicalNat v v'
```

C.5 Proof of index expression thinning properties

```
thin : Ix (ictx1 ++ ictx) t
  -> (ictx2 : IxCtx)
  -> Ix  (ictx1 ++ ictx2 ++ ictx) t
thin (S i) ictx2 = S (thin i ictx2)
thin Z ictx2 = Z
thin (Var elt) ictx2 = Var (thin elt ictx2)
thin Nil ictx2 = Nil
thin (i :: is) ictx2 = thin i ictx2 :: thin is ictx2
thin {ictx1=ictx1} (CaseList i1 hd tl i2 i3) ictx2 =
  CaseList (thin i1 ictx2)
    hd tl (thin {ictx1=(hd, TyNat) :: (tl, TyList) :: ictx1}
                i2 ictx2)
    (thin i3 ictx2)
thin {ictx1=ictx1} (ElimList {k=k} i1 acc x i2 i3) ictx2 =
  ElimList (thin i1 ictx2)
    acc x (thin {ictx1=(acc, k) :: (x, TyNat) :: ictx1} i2 ictx2)
    (thin i3 ictx2)
thin {ictx1=ictx1} (ElimNat {k=k} i1 acc x i2 i3) ictx2 =
  ElimNat (thin i1 ictx2)
    acc x (thin {ictx1=(acc, k) :: (x, TyNat) :: ictx1} i2 ictx2)
    (thin i3 ictx2)
```

Lemma C.5.0.1 (Thinning respects evaluation)**.**

```
  thinRespectsEval
    : (i : Ix (ictx1 ++ ictx) k)
   -> (ixEnv1 : IxEnv ictx1)
   -> (ixEnv : IxEnv ictx)
   -> (ixEnv2 : IxEnv ictx2)
   -> eval (thin i ictx2) (ixEnv1 <++> ixEnv2 <++> ixEnv) =
      eval i (ixEnv1 <++> ixEnv)
```

.

Proof.
```
thinRespectsEval (S i) ixEnv1 ixEnv ixEnv2 =
  cong (thinRespectsEval i ixEnv1 ixEnv ixEnv2)
thinRespectsEval Z ixEnv1 ixEnv ixEnv2 = Refl
thinRespectsEval (Var elt) ixEnv1 ixEnv ixEnv2 =
  thinRespectsLookup elt ixEnv1 ixEnv ixEnv2
thinRespectsEval [] ixEnv1 ixEnv ixEnv2 = Refl
thinRespectsEval (i :: is) ixEnv1 ixEnv ixEnv2 =
  consCong (thinRespectsEval i ixEnv1 ixEnv ixEnv2)
           (thinRespectsEval is ixEnv1 ixEnv ixEnv2)
thinRespectsEval {ictx2=ictx2} (CaseList i1 x xs i2 i3)
                 ixEnv1 ixEnv ixEnv2 =
  listDestructCong
    (\n, ns => eval (thin i2 ictx2)
                    ((n :: ns :: ixEnv1) <++>
                     ixEnv2 <++> ixEnv))
    (\n, ns => eval i2 ((n :: ns :: ixEnv1) <++> ixEnv))
    (eval (thin i3 ictx2) (ixEnv1 <++> ixEnv2 <++> ixEnv))
    (eval i3 (ixEnv1 <++> ixEnv))
    (thinRespectsEval i1 ixEnv1 ixEnv ixEnv2)
    (\n, ns =>
      thinRespectsEval i2 (n :: ns :: ixEnv1) ixEnv ixEnv2)
    (thinRespectsEval i3 ixEnv1 ixEnv ixEnv2)
thinRespectsEval {ictx2=ictx2} (ElimList i1 acc x i2 i3)
                 ixEnv1 ixEnv ixEnv2 =
  listFoldCong
    (\v, acc => eval (thin i2 ictx2)
                     ((acc :: v :: ixEnv1) <++>
                      ixEnv2 <++> ixEnv))
    (\v, acc => eval i2 ((acc :: v :: ixEnv1) <++> ixEnv))
    (eval (thin i3 ictx2) (ixEnv1 <++> ixEnv2 <++> ixEnv))
    (eval i3 (ixEnv1 <++> ixEnv))
    (thinRespectsEval i1 ixEnv1 ixEnv ixEnv2)
    (\v, acc =>
      thinRespectsEval i2 (acc :: v :: ixEnv1) ixEnv ixEnv2)
    (thinRespectsEval i3 ixEnv1 ixEnv ixEnv2)
thinRespectsEval {ictx2=ictx2} (ElimNat i1 acc x i2 i3)
                 ixEnv1 ixEnv ixEnv2 =
  natFoldCong
    (\v, acc => eval (thin i2 ictx2)
                     ((acc :: v :: ixEnv1) <++>
                      ixEnv2 <++> ixEnv))
    (\v, acc => eval i2 ((acc :: v :: ixEnv1) <++> ixEnv))
    (eval (thin i3 ictx2) (ixEnv1 <++> ixEnv2 <++> ixEnv))
    (eval i3 (ixEnv1 <++> ixEnv))
    (thinRespectsEval i1 ixEnv1 ixEnv ixEnv2)
```

```
(\v, acc =>
  thinRespectsEval i2 (acc :: v :: ixEnv1)
                   ixEnv ixEnv2)
(thinRespectsEval i3 ixEnv1 ixEnv ixEnv2)
```

■

Appendix D

Specification and compilation of $\mathscr{S}$

```
mutual
  data StmtBranch : (ictx : IxCtx)
                 -> (facts : IxConstraints ictx)
                 -> (i, j : Ix ictx t)
                 -> IxConstraints ictx3
                 -> Ctx ictx3 -> (i3 : Ix ictx3 t3)
                 -> Type where
    Reachable : IsReachable facts i j
             -> Stmt facts3 ctx3 t3 i3
             -> StmtBranch ictx facts i j facts3 ctx3 i3
    Unreachable : NotReachable facts i j
               -> (ctx3 : Ctx ictx3)
               -> (i3 : Ix ictx3 t3)
               -> StmtBranch ictx facts i j facts3 ctx3 i3

  data Stmt : IxConstraintsictx
           -> Ctxictx
           -> (t : Ty)
           -> Ix ictx t
           -> Type where
    Atomic : AExp ctx t i -> Stmt facts ctx t i
    Rewrite : CanUnifyIx facts i j
           -> Stmt facts ctx t i
           -> Stmt facts ctx t j
    Let : (x : Name)
       -> {ctx : Ctx ictx}
       -> Stmt facts ctx t1 i1
```

```
      -> Stmt facts ((x, (t1 ** i1)) :: ctx) t2 i2
      -> Stmt facts ctx t2 i2
Nil : Stmt facts ctx TyList []
(::) : Stmt facts ctx TyNat i
    -> Stmt facts ctx TyList is
    -> Stmt facts ctx TyList (i :: is)
CaseList : {ctx : Ctx ictx}
        -> Stmt facts ctx TyList i1
        -> (hd, tl : Name)
        -> StmtBranch
             ([(hd, TyNat), (tl, TyList)] ++ ictx)
             (weakenConstraints facts
                [(hd, TyNat), (tl, TyList)])
             (weaken i1 [(hd, TyNat), (tl, TyList)])
             (Var Here :: Var (There Here))
             ((RequireEq
                 (Var Here :: Var (There Here))
                 (weaken i1 [_, _])) ::
              weakenConstraints facts [_, _])
             ((hd, (TyNat ** Var Here)) ::
              (tl, (TyList ** Var (There Here))) ::
              weaken ctx [(hd, TyNat), (tl, TyList)])
             i2
        -> StmtBranch
             ictx facts i1 []
             (RequireEq [] i1 :: facts) ctx i3
        -> Stmt facts ctx t' (CaseList i1 hd tl i2 i3)
ElimList : {ctx : Ctx ictx}
        -> Stmt facts ctx TyList i1
        -> (x, acc : Name)
        -> Stmt (weakenConstraints facts [_,_])
                ((acc, (t ** Var Here)) ::
                 (x, (TyNat ** Var (There Here))) ::
                 weaken ctx [_,_])
                t
                i2
        -> Stmt facts ctx t i3
        -> Stmt facts ctx t (ElimList i1 acc x i2 i3)
ElimNat : {ctx : Ctx ictx}
       -> {i3 : Ix ictx t}
       -> Stmt facts ctx TyNat i1
       -> (x, acc : Name)
       -> {i2 : Ix ((acc, t) :: (x, TyNat) :: ictx) t}
       -> Stmt (weakenConstraints facts
                  [(acc,t), (x,TyNat)])
```

```
                ((acc, (t ** Var Here)) ::
                 (x, (TyNat ** Var (There Here))) ::
                 weaken ctx [(acc,t),(x,TyNat)])
                t
                i2
          -> Stmt facts ctx t i3
          -> Stmt facts ctx t (ElimNat i1 acc x i2 i3)
```

D.1 Compiling from $\mathscr{S}$ to machine code

Listing D.1: Concatenation of well-formed compile environments

```
appendCEnv : CompileEnv ctx2 mctx
          -> CompileEnv ctx1 mctx
          -> CompileEnv (ctx2 ++ ctx1) mctx
appendCEnv EmptyEnv cenv1 = cenv1
appendCEnv (UExtend x t i v cenv2) cenv1 =
  UExtend x t i v (cenv2 `appendCEnv` cenv1)

compile : {ctx : Ctx ictx}
     -> CompileEnv ctx mctx
     -> Stmt facts ctx t i
     -> ((ictx' : IxCtx) ->
         (ctx' : Ctx (ictx' ++ ictx)) ->
         CompileEnv ctx' mctx ->
         MVal mctx t ->
         Asm mctx lctx t')
     -> Asm mctx lctx t'
compile cenv (Atomic a) k = compileAtomic cenv a (k [])
compile cenv (Rewrite iToJ s) k = compile cenv s k
compile {ictx=ictx} cenv (Let {i1=i1} {t1=t1} x s1 s2) k =
  compile cenv s1 (\ictx1, ctx1, cenv1, v1 =>
  compile (UExtend x t1 i1 v1 cenv) s2 (\ictx2, ctx2, cenv2, v2 =>
  k (ictx2 ++ ictx1)
    (replace (appendAssociative ictx2 ictx1 ictx)
             (thin ctx2 ictx1) ++
     replace (appendAssociative ictx2 ictx1 ictx)
             (weaken ctx1 ictx2))
    (cenvIxCtx (appendAssociative ictx2 ictx1 ictx)
               (thinCEnvCtx cenv2 ictx1) `appendCEnv`
     cenvIxCtx (appendAssociative ictx2 ictx1 ictx)
               (weakenCEnvCtx cenv1 ictx2))
    v2))
compile cenv [] k =
  AllocTagged ConsTyList TagListNil noNilPositions
```

```
              (\v => k [] [] EmptyEnv (Var v))
  where noNilPositions : (posTy : Ty)
                      -> ConsPos TagListNil posTy
                      -> MVal mctx posTy
        noNilPositions posTy ConsPosHead impossible
        noNilPositions posTy ConsPosTail impossible
compile {ictx=ictx} cenv (s1 :: s2) k =
  compile cenv s1 (\ictx1, ctx1, cenv1, v1 =>
  compile cenv s2 (\ictx2, ctx2, cenv2, v2 =>
  AllocTagged ConsTyList TagListCons (mkcons v1 v2) (\v3 =>
  k (ictx2 ++ ictx1)
    (replace (appendAssociative ictx2 ictx1 ictx)
             (thin ctx2 ictx1) ++
     replace (appendAssociative ictx2 ictx1 ictx)
             (weaken ctx1 ictx2))
    (cenvIxCtx (appendAssociative ictx2 ictx1 ictx)
               (thinCEnvCtx cenv2 ictx1) `appendCEnv`
     cenvIxCtx (appendAssociative ictx2 ictx1 ictx)
               (weakenCEnvCtx cenv1 ictx2))
    (Var v3))))
  where mkcons : MVal mctx TyNat -> MVal mctx TyList
              -> (posTy : Ty) -> ConsPos TagListCons posTy
              -> MVal mctx posTy
        mkcons hd tl TyNat  ConsPosHead = hd
        mkcons hd tl TyList ConsPosTail = tl
compile {t=TyList} {t'=t'} cenv
  (CaseList {ictx=ictx} s1 hd tl (Reachable iToJ2 s2)
                                 (Reachable iToJ3 s3)) k =
  compile cenv s1 (\ictx1, ctx1, cenv1, v1 =>
    let v1' = unwrapVar v1 in
    GetTag v1' (\isConsTag =>
    Jz isConsTag
       (compile cenv s3 (\ictx3, ctx3, cenv3, v3 =>
        k (ictx3 ++ ictx1)
          (replace (appendAssociative ictx3 ictx1 ictx)
                   (thin ctx3 ictx1) ++
           replace (appendAssociative ictx3 ictx1 ictx)
                   (weaken ctx1 ictx3))
          (cenvIxCtx (appendAssociative ictx3 ictx1 ictx)
                     (thinCEnvCtx cenv3 ictx1) `appendCEnv`
           cenvIxCtx (appendAssociative ictx3 ictx1 ictx)
                     (weakenCEnvCtx cenv1 ictx3))
          v3))
       (GetConsPos v1' ConsPosHead (\v1Hd =>
        GetConsPos v1' ConsPosTail (\v1Tl =>
```

```
        compile {ictx=(hd,TyNat) :: (tl,TyList) :: ictx}
                (UExtend hd _ _ v1Hd
                  (UExtend tl _ _ v1Tl
                    (weakenCEnvCtx cenv [_,_])))
                s2
                (\ictx2, ctx2, cenv2, v2 =>
        k (ictx2 ++ ictx1 ++ [(hd, TyNat), (tl, TyList)])
          (replace (shuffleAssoc ictx2 ictx1
                                 [(hd, TyNat), (tl, TyList)] ictx)
                   (thin ctx2 ictx1) ++
           replace (shuffleAssoc ictx2 ictx1
                                 [(hd, TyNat), (tl, TyList)] ictx)
                   (weaken (thin ctx1
                                 [(hd, TyNat), (tl, TyList)])
                           ictx2))
          (cenvIxCtx
             (shuffleAssoc ictx2 ictx1
                           [(hd, TyNat), (tl, TyList)] ictx)
             (thinCEnvCtx cenv2 ictx1) `appendCEnv`
           cenvIxCtx
             (shuffleAssoc ictx2 ictx1 [(hd, TyNat), (tl, TyList)] ictx)
             (weakenCEnvCtx (thinCEnvCtx cenv1
                                         [(hd, TyNat), (tl, TyList)])
                            ictx2))
          v2))))))
  where shuffleAssoc : (xs3, xs2, xs1, xs : IxCtx)
                    -> xs3 ++ xs2 ++ xs1 ++ xs =
                       (xs3 ++ xs2 ++ xs1) ++ xs
        shuffleAssoc xs3 xs2 xs1 xs =
          replace {P=\x => xs3 ++ xs2 ++ xs1 ++ xs = x}
            (appendAssociative xs3 (xs2 ++ xs1) xs)
            (replace {P=\x => xs3 ++ xs2 ++ xs1 ++ xs = xs3 ++ x}
               (appendAssociative xs2 xs1 xs)
               (Refl {x=xs3 ++ (xs2 ++ (xs1 ++ xs))}))
compile {t=TyList} {t'=t'} cenv
  (CaseList {ictx=ictx} s1
     hd tl (Reachable iToJ2 s2)
     (Unreachable (NoReach iNoNil) _ _)) k =
  compile cenv s1 (\ictx1, ctx1, cenv1, v1 =>
    let v1' = unwrapVar v1 in
    GetConsPos v1' ConsPosHead (\v1Hd =>
    GetConsPos v1' ConsPosTail (\v1Tl =>
    compile (UExtend hd _ _ v1Hd
              (UExtend tl _ _ v1Tl
                (weakenCEnvCtx cenv [_,_])))
```

```
           s2
           (\ictx2, ctx2, cenv2, v2 =>
    k (ictx2 ++ ictx1 ++ [(hd, TyNat), (tl, TyList)])
      (replace (shuffleAssoc ictx2 ictx1
                             [(hd, TyNat), (tl, TyList)] ictx)
               (thin ctx2 ictx1) ++
       replace (shuffleAssoc ictx2 ictx1
                             [(hd, TyNat), (tl, TyList)] ictx)
               (weaken (thin ctx1
                             [(hd, TyNat), (tl, TyList)])
                       ictx2))
      (cenvIxCtx
         (shuffleAssoc ictx2 ictx1 [(hd, TyNat), (tl, TyList)] ictx)
         (thinCEnvCtx cenv2 ictx1) `appendCEnv`
       cenvIxCtx
         (shuffleAssoc ictx2 ictx1 [(hd, TyNat), (tl, TyList)] ictx)
         (weakenCEnvCtx (thinCEnvCtx cenv1
                                     [(hd, TyNat), (tl, TyList)])
                        ictx2))
      v2))))
  where shuffleAssoc : (xs3, xs2, xs1, xs : IxCtx)
                    -> xs3 ++ xs2 ++ xs1 ++ xs =
                       (xs3 ++ xs2 ++ xs1) ++ xs
        shuffleAssoc xs3 xs2 xs1 xs =
          replace {P=\x => xs3 ++ xs2 ++ xs1 ++ xs = x}
            (appendAssociative xs3 (xs2 ++ xs1) xs)
            (replace {P=\x => xs3 ++ xs2 ++ xs1 ++ xs = xs3 ++ x}
               (appendAssociative xs2 xs1 xs)
               (Refl {x=xs3 ++ (xs2 ++ (xs1 ++ xs))}))
compile {t=TyList} {t'=t'} cenv
  (CaseList {ictx=ictx} s1 hd tl (Unreachable (NoReach iNoCons) _ _)
                                 (Reachable iToJ3 s3)) k =
  compile cenv s3 k
compile {ictx=ictx} cenv (ElimList s1 x acc s2 s3) k =
  compile cenv s1 (\ictx1, ctx1, cenv1, vxs0 =>
  compile cenv s3 (\ictx3, ctx3, cenv3, vacc0 =>
  PLabel _ _ _ [vacc0, vxs0] (\l, [vacc, vxs] =>
    GetTag vxs (\isConsTag =>
    Jz isConsTag
       (k (ictx3 ++ ictx1)
          (replace (appendAssociative ictx3 ictx1 ictx)
                   (thin ctx3 ictx1) ++
           replace (appendAssociative ictx3 ictx1 ictx)
                   (weaken ctx1 ictx3))
        (cenvIxCtx (appendAssociative ictx3 ictx1 ictx)
```

```
                  (thinCEnvCtx cenv3 ictx1) `appendCEnv`
           cenvIxCtx (appendAssociative ictx3 ictx1 ictx)
                  (weakenCEnvCtx cenv1 ictx3))
         (Var vacc))
       (GetConsPos vxs ConsPosHead (\vx =>
        compile (UExtend acc _ _ (Var vacc)
                  (UExtend x _ _ vx
                    (weakenCEnvCtx cenv [_,_])))
                s2
                (\ictx2, ctx2, cenv2 , vacc' =>
          GetConsPos vxs ConsPosTail (\vxs' =>
          PJmp l [vacc', vxs'])))))))))
compile {ictx=ictx} cenv (ElimNat s1 x acc s2 s3) k =
  compile cenv s1 (\ictx1, ctx1, cenv1, vx0 =>
  compile cenv s3 (\ictx3, ctx3, cenv3, vacc0 =>
  PLabel _ _ _ [vacc0, vx0] (\l, [vacc, vx] =>
    Jz (Var vx)
       (k (ictx3 ++ ictx1)
          (replace (appendAssociative ictx3 ictx1 ictx)
                   (thin ctx3 ictx1) ++
           replace (appendAssociative ictx3 ictx1 ictx)
                   (weaken ctx1 ictx3))
          (cenvIxCtx (appendAssociative ictx3 ictx1 ictx)
                     (thinCEnvCtx cenv3 ictx1) `appendCEnv`
           cenvIxCtx (appendAssociative ictx3 ictx1 ictx)
                     (weakenCEnvCtx cenv1 ictx3))
          (Var vacc))
       (compile (UExtend acc _ _ (Var vacc)
                  (UExtend x _ _ (Var vx)
                    (weakenCEnvCtx cenv [_,_])))
                s2
                (\ictx2, ctx2, cenv2 , vacc' =>
          Dec (Var vx) (\vx' =>
          PJmp l [vacc', vx'])))))))
```

Listing D.2: Evaluation with equality

```
record RevealEval (f : Ix ictx t -> IxEnv ictx -> EvalTy t)
                  (i : Ix ictx t)
                  (ixEnv : IxEnv ictx)
                  (v : EvalTy t) where
  constructor EvalIs
  evalIs : f i ixEnv = v

inspectEval : (i : Ix ictx t)
           -> (ixEnv : IxEnv ictx)
```

```
          -> RevealEval eval i ixEnv (eval i ixEnv)
inspectEval i ixEnv = EvalIs Refl
```

Listing D.3: Fold substitution rule

```
natFoldFSubst : ((n : Nat) -> (y : a) -> f n y = g n y)
             -> (m : Nat) -> (x : a)
             -> natFold f x m = natFold g x m
natFoldFSubst fEg Z x = Refl
natFoldFSubst {f = f}{g = g} fEg (S m) x =
  replace {P=\v => f m (natFold f x m) = g m v}
    (natFoldFSubst fEg m x)
    (fEg m (natFold f x m))
```

Listing D.4: List fold substitution rule

```
listFoldFSubst : ((n : Nat) -> (y : a) -> f n y = g n y)
              -> (xs : List Nat) -> (v : a)
              -> listFold f v xs = listFold g v xs
listFoldFSubst fEg [] v = Refl
listFoldFSubst {f = f}{g = g} fEg (x :: xs) v =
  replace {P=\z => f x (listFold f v xs) = g x z}
    (listFoldFSubst fEg xs v)
    (fEg x (listFold f v xs))
```

Listing D.5: Abstract machine evaluation function

```
eval : Asm EvalTy EvalLabel t -> (clk : Nat) -> Maybe (EvalTy t)
eval _ Z = Nothing
eval (AllocTagged ConsTyList TagListNil vf k) (S clk) =
  eval (k []) clk
eval (AllocTagged ConsTyList TagListCons vf k) (S clk) =
  let v  = evalMVal (vf TyNat ConsPosHead)
      vs = evalMVal (vf TyList ConsPosTail)
  in eval (k (v :: vs)) clk
eval (GetConsPos {t=TyList} [] ConsPosHead k) (S clk) = Nothing
eval (GetConsPos {t=TyList} (x :: xs) ConsPosHead k) (S clk) =
  eval (k (Var x)) clk
eval (GetConsPos {t=TyList} [] ConsPosTail k) (S clk) = Nothing
eval (GetConsPos {t=TyList} (x :: xs) ConsPosTail k) (S clk) =
  eval (k (Var xs)) clk
eval (GetTag {t=TyList} [] k) (S clk) =
  eval (k (Lit (NLit (enumConsTag TagListNil)))) clk
eval (GetTag {t=TyList} (_ :: _) k) (S clk) =
  eval (k (Lit (NLit (enumConsTag TagListCons)))) clk
eval (Jz v p1 p2) (S clk) = case evalMVal v of
```

```
  Z => eval p1 clk
  S n => eval p2 clk
eval (Dec v k) (S clk) = eval (k (Var (pred (evalMVal v)))) clk
eval (Add v1 v2 k) (S clk) =
  eval (k (Var (evalMVal v1 + evalMVal v2))) clk
eval {t=t} (PJmp {t=t} k args) (S clk) = k (evalArgs args)
eval (PLabel n ts t args k) (S clk) =
  let vs = evalArgs args
  in bfix (\vs', rec => eval (k rec vs') clk) clk vs
  where bfix : (All EvalTy ts ->
                (All EvalTy ts -> Maybe (EvalTy t)) ->
                Maybe (EvalTy t))
            -> Nat
            -> All EvalTy ts
            -> Maybe (EvalTy t)
        bfix f Z     vs = Nothing
        bfix f (S n) vs = f vs (bfix f n)
```

Appendix E

Equivalence of well-formed De Bruijn indices and context membership

```
data WfElem : Nat -> List a -> Type where
  Here : WfElem Z (x :: xs)
  There : WfElem n xs -> WfElem (S n) (y :: xs)

wflookup : (ctx : List a) -> WfElem n ctx -> a
wflookup [] elt impossible
wflookup (x :: xs) Here        = x
wflookup (y :: xs) (There elt) = wflookup xs elt

to : (n : Nat) -> .(okn : WfElem n ctx)
  -> Elem (wflookup ctx okn) ctx
to Z     Here        = Here
to (S n) (There elt) = There (to n elt)

from : Elem x ctx
    -> (n : Nat ** okn : WfElem n ctx **
        wflookup ctx okn = x)
from Here = (Z ** Here ** Refl)
from (There elt) with (from elt)
  from (There elt) | (n ** okn ** pf) =
    (S n ** There okn ** pf)
```

Appendix F

Definition of the IfNil relation

$$\frac{C_1 = C'}{\text{IfNil}(x :: xs, C_1, C_2, C')} \qquad \frac{C_2 = C'}{\text{IfNil}([], C_1, C_2, C')}$$

Bibliography

[AAG05] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. "Containers: Constructing strictly positive types". In: *Theor. Comput. Sci.* 342.1 (2005), pp. 3–27. DOI: 10.1016/j.tcs.2005.06.002.

[Abd13] Moez A. AbdelGawad. "An Overview of Nominal-Typing versus Structural-Typing in Object-Oriented Programming". In: *CoRR* abs/1309.2348 (2013). arXiv: 1309.2348. URL: http://arxiv.org/abs/1309.2348.

[AC99] Lennart Augustsson and Magnus Carlsson. "An exercise in dependent types: A well-typed interpreter". In: *In Workshop on Dependent Types in Programming, Gothenburg*. 1999.

[Alb+07] Elvira Albert et al. "COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode". In: *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*. 2007, pp. 113–132. DOI: 10.1007/978-3-540-92188-2_5.

[Alt+15] Thorsten Altenkirch et al. "Indexed containers". In: *J. Funct. Program.* 25 (2015). DOI: 10.1017/S095679681500009X.

[ALY09] Robert Atkey, Sam Lindley, and Jeremy Yallop. "Unembedding domain-specific languages". In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*. 2009, pp. 37–48. DOI: 10.1145/1596638.1596644.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational equality, now!" In: *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*. 2007, pp. 57–68. DOI: 10.1145/1292597.1292608.

[AP11] Andreas Abel and Brigitte Pientka. "Higher-Order Dynamic Pattern Unification for Dependent Types and Records". In: *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*. 2011, pp. 10–26. DOI: 10.1007/978-3-642-21691-6_5.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN: 0-521-41695-7.

[App98] Andrew W. Appel. "SSA is Functional Programming". In: *SIGPLAN Notices* 33.4 (1998), pp. 17–20. DOI: 10.1145/278283.278285.

[ARS94] Lennart Augustsson, Mikael Rittri, and Dan Synek. "On Generating unique Names". In: *J. Funct. Program.* 4.1 (1994), pp. 117–123. DOI: 10.1017/S0956796800000988.

[AS13] Johan Ankner and Josef Svenningsson. "An EDSL approach to high performance Haskell programming". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. 2013, pp. 1–12. DOI: 10.1145/2503778.2503789.

[Asp+06] Andrea Asperti et al. "Crafting a Proof Assistant". In: *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*. 2006, pp. 18–32. DOI: 10.1007/978-3-540-74464-1_2.

[Asp95] David Aspinall. "Subtyping with singleton types". In: *Computer Science Logic*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–15. ISBN: 978-3-540-49404-1.

[Ass16] Digital Asset. *The Digital Asset Platform*. Tech. rep. Digital Assets, 2016. URL: https://hub.digitalasset.com/hubfs/Documents/Digital%20Asset%20Platform%20-%20Non-technical%20White%20Paper.pdf.

[Atk12] Robert Atkey. "Relational Parametricity for Higher Kinds". In: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*. 2012, pp. 46–61. DOI: 10.4230/LIPIcs.CSL.2012.46.

[Atk15] Robert Atkey. "An Algebraic Approach To Typechecking and Elaboration". Higher Order Programming with Effects (HOPE) 2015. Feb. 2015. URL: https://bentnib.org/docs/algebraic-typechecking-20150218.pdf.

[Aug98] Lennart Augustsson. "Cayenne - a Language with Dependent Types". In: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. 1998, pp. 239–250. DOI: 10.1145/289423.289451.

[Aug99] Lennart Augustsson. *Equality proofs in Cayenne*. Tech. rep. 1999.

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993. ISBN: 978-0-13-285792-5.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. "Detecting Equality of Variables in Programs". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. 1988, pp. 1–11. DOI: 10.1145/73560.73561.

[Axe+10] Emil Axelsson et al. "The Design and Implementation of Feldspar - An Embedded Language for Digital Signal Processing". In: *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*. 2010, pp. 121–136. DOI: 10.1007/978-3-642-24276-2_8.

[Ayd+05] Brian E. Aydemir et al. "Mechanized Metatheory for the Masses: The PoplMark Challenge". In: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*. 2005, pp. 50–65. DOI: 10.1007/11541868_4.

[Ayd+08] Brian Aydemir et al. "Engineering Formal Metatheory". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 3–15. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328443.

[Baa+10] Christiaan Baaij et al. "CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell". In: *13th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2010, 1-3 September 2010, Lille, France*. 2010, pp. 714–721. DOI: 10.1109/DSD.2010.21.

[Bar91] Henk Barendregt. "Introduction to Generalized Type Systems". In: *J. Funct. Program.* 1.2 (1991), pp. 125–154.

[BC05] Ana Bove and Venanzio Capretta. "Modelling general recursion in type theory". In: *Mathematical Structures in Computer Science* 15.4 (2005), pp. 671–708. DOI: 10.1017/S0960129505004822.

[BH05] Edwin Brady and Kevin Hammond. "A Dependently Typed Framework for Static Analysis of Program Execution Costs". In: *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers*. 2005, pp. 74–90. DOI: 10.1007/11964681_5.

[BM96] Richard S. Bird and Oege de Moor. "The algebra of programming". In: *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*. 1996, pp. 167–203.

[BMM03] Edwin Brady, Conor McBride, and James McKinna. "Inductive Families Need Not Store Their Indices". In: *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*. 2003, pp. 115–129. DOI: 10.1007/978-3-540-24849-1_8. URL: https://doi.org/10.1007/978-3-540-24849-1%5C_8.

[Boe10] Mathieu Boespflug. "Conversion by Evaluation". In: *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*. 2010, pp. 58–72. DOI: 10.1007/978-3-642-11503-5_7.

[BP12] Andrej Bauer and Matija Pretnar. "Programming with Algebraic Effects and Handlers". In: *CoRR* abs/1203.1539 (2012). arXiv: 1203.1539. URL: http://arxiv.org/abs/1203.1539.

[BP99] Richard S. Bird and Ross Paterson. "De Bruijn Notation as a Nested Datatype". In: *J. Funct. Program.* 9.1 (1999), pp. 77–91. URL: http://journals.cambridge.org/action/displayAbstract?aid=44239.

[Bra05a] Edwin Brady. "Practical Implementation of a Dependently Typed Functional Programming Language". PhD thesis. Computer Science Dept., University of Durham, England, 2005.

[Bra05b] Edwin Brady. "Practical Implementation of a Dependently Typed Functional Programming Language". PhD thesis. Computer Science Dept., University of Durham, England, 2005, pp. 149–166.

[Bra13a] Edwin Brady. "First-class Type-safe Reflection in Idris". Workshop on Dependently-Typed Programming DTP 2013. Sept. 2013. URL: https://www.seas.upenn.edu/~sweirich/dtp13/talks/brady.pdf.

[Bra13b] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *J. Funct. Program.* 23.5 (2013), pp. 552–593. DOI: 10.1017/S095679681300018X.

[Bra14a] Edwin Brady. "Idris: Implementing a Dependently Typed Programming Language". In: *Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP '14, Vienna, Austria, July 17, 2014*. 2014, 2: 1. DOI: 10.1145/2631172.2631174.

[Bra14b] Edwin Brady. "Resource-Dependent Algebraic Effects". In: *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*. 2014, pp. 18–33. DOI: 10.1007/978-3-319-14675-1_2.

[Bra16] Edwin Brady. *Type-driven Development With Idris*. Manning, 2016. ISBN: 9781617293023.

[Bra18] Edwin Brady. *Blodwen*. Tech. rep. 2018. URL: https://github.com/edwinb/Blodwen.

[Bru72] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0. URL: http://www.sciencedirect.com/science/article/pii/1385725872900340.

[BW97] Bruno Barras and Benjamin Werner. *Coq in Coq*. Tech. rep. INRIA, Rocquencourt, 1997.

[Cap05] Venanzio Capretta. "General recursion via coinductive types". In: *Logical Methods in Computer Science* 1.2 (2005). DOI: 10.2168/LMCS-1(2:1)2005.

[CB16] David R. Christiansen and Edwin Brady. "Elaborator reflection: extending Idris in Idris". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 2016, pp. 284–297. DOI: 10.1145/2951913.2951932.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. "Pattern matching without K". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. 2014, pp. 257–268. DOI: 10.1145/2628136.2628139.

[CF67] Haskell B. Curry and Robert Feys. "Combinatory logic". In: *Journal of Symbolic Logic* 32.2 (1967). Ed. by J. Barkley Rosser, pp. 267–268. DOI: 10.1017/S0022481200114203.

[CH88] Thierry Coquand and Gérard P. Huet. "The Calculus of Constructions". In: *Inf. Comput.* 76.2/3 (1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3.

[Cha+10] James Chapman et al. "The gentle art of levitation". In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 2010, pp. 3–14. DOI: 10.1145/1863543.1863547.

[Cha12] Arthur Charguéraud. "The Locally Nameless Representation". In: *J. Autom. Reasoning* 49.3 (2012), pp. 363–408. DOI: 10.1007/s10817-011-9225-2.

[Chl08] Adam Chlipala. "Parametric higher-order abstract syntax for mechanized semantics". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 2008, pp. 143–156. DOI: 10.1145/1411204.1411226.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN: 978-0-262-02665-9. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.

[Chr14] David Raymond Christiansen. "Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection". In: *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1-3, 2014*. 2014, 1:1–1:9. DOI: 10.1145/2746325.2746326.

[Chu40] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *J. Symb. Log.* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941. ISBN: 9780691083940.

[CKZ03] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. "A Functional Perspective on SSA Optimisation Algorithms". In: *Electr. Notes Theor. Comput. Sci.* 82.2 (2003), pp. 347–361. DOI: 10.1016/S1571-0661(05)82596-4.

[Com10] The Haskell Committee. *Haskell 2010 Language Report*. Tech. rep. 2010. URL: https://www.haskell.org/onlinereport/haskell2010/.

[Com99] The C Standards Committee. *Programming languages — C*. Standard. International Organization for Standardization, 1999, pp. 42–49.

[Coq86] Thierry Coquand. "An Analysis of Girard's Paradox". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. 1986, pp. 227–236.

[Coq92] Thierry Coquand. "Pattern matching with dependent types". In: *Proceedings of the Workshop on Types for Proofs and Programs*. 1992, pp. 71–83.

[Cor16] Intel Corporation. *Intel (R) 64 and IA-32 Architectures Optimization Reference Manual*. June 2016, pp. 89–97. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[CP90] Thierry Coquand and Christine Paulin. "Inductively Defined Types". In: *Proceedings of the International Conference on Computer Logic*. COLOG '88. London, UK, UK: Springer-Verlag, 1990, pp. 50–66. ISBN: 3-540-52335-9. URL: http://dl.acm.org/citation.cfm?id=646125.758641.

[Cra03] Karl Crary. "Toward a foundational typed assembly language". In: *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*. 2003, pp. 198–212. DOI: 10.1145/640128.604149.

[CRK05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers - where the Kernel meets the hardware (3. ed.)* O'Reilly, 2005. ISBN: 978-0-596-00590-0. URL: http://www.oreilly.de/catalog/linuxdrive3/.

[CW00] Karl Crary and Stephanie Weirich. "Resource Bound Certification". In: *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*. 2000, pp. 184–198. DOI: 10.1145/325694.325716.

[CW99] Karl Crary and Stephanie Weirich. "Flexible Type Analysis". In: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*. 1999, pp. 233–248. DOI: 10.1145/317636.317906.

[Dan12] Nils Anders Danielsson. "Operational semantics using the partiality monad". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. 2012, pp. 127–138. DOI: 10.1145/2364527.2364546.

[Del00] David Delahaye. "A Tactic Language for the System Coq". In: *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*. 2000, pp. 85–95. DOI: 10.1007/3-540-44404-1_7.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (July 2008), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410.

[Dyb94] Peter Dybjer. "Inductive families". In: *Formal Aspects of Computing* 6.4 (July 1994), pp. 440–465. ISSN: 1433-299X. DOI: 10.1007/BF01211308.

[Ell+15] Trevor Elliott et al. "Guilt free ivory". In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 2015, pp. 189–200. DOI: 10.1145/2804302.2804318.

[FH92] Matthias Felleisen and Robert Hieb. "The Revised Report on the Syntactic Theories of Sequential Control and State". In: *Theor. Comput. Sci.* 103.2 (1992), pp. 235–271. DOI: 10.1016/0304-3975(92)90014-7.

[Fla+93] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*. 1993, pp. 237–247. DOI: 10.1145/155090.155113.

[FMA06] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. "Linear Regions Are All You Need". In: *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*. 2006, pp. 7–21. DOI: 10.1007/11693024_2.

[Gil+09] Andy Gill et al. "Introducing Kansas Lava". In: *Proceedings of the Symposium on Implementation and Application of Functional Languages*. Vol. 6041. LNCS. Springer-Verlag, Sept. 2009.

[Gir72] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Université Paris 7, 1972.

[GL02] Benjamin Grégoire and Xavier Leroy. "A compiled implementation of strong reduction". In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*. 2002, pp. 235–246. DOI: 10.1145/581478.581501.

[GM05] Benjamin Grégoire and Assia Mahboubi. "Proving Equalities in a Commutative Ring Done Right in Coq". In: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*. 2005, pp. 98–113. DOI: 10.1007/11541868_7.

[GM12] Adam Gundry and Conor McBride. "A tutorial implementation of dynamic pattern unification". preprint on webpage at adam.gundry.co.uk/pub/pattern-unify/pattern-unification-2012-07-10.pdf. July 2012.

[Gol84] Robert Goldblatt. *Topoi: The Categorial Analysis of Logic*. Revised ed. (26 May 2006). Dover Publications Inc., 1984, p. 576.

[Gon+13] Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.

[Gon07] Georges Gonthier. "The Four Colour Theorem: Engineering of a Formal Proof". In: *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*. 2007, p. 333. DOI: 10.1007/978-3-540-87827-8_28.

[Hab71] A. Nico Habermann. *Introduction to ALGOL 60 for those who have used other programming languages*. Tech. rep. Available at repository.cmu.edu/cgi/viewcontent.cgi?article=2981&context=compsci. School of Computer Science Department, Carnegie Mellon University, 1971.

[Har12] Professor Robert Harper. *Practical Foundations for Programming Languages*. New York, NY, USA: Cambridge University Press, 2012.

[Her05] Hugo Herbelin. *Type inference with algebraic universes in the Calculus of Inductive Constructions*. Tech. rep. 2005.

[Hof95] Martin Hofmann. "Conservativity of Equality Reflection over Intensional Type Theory". In: *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*. 1995, pp. 153–164. DOI: 10.1007/3-540-61780-9_68.

[Hol14] ARM Holdings. *ARM (R) NEON (tm) Intrinsics Reference*. Sept. 2014, p. 344. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0073a/IHI0073A_arm_neon_intrinsics_ref.pdf.

[How80] W. A. Howard. "The formulae-as-types notion of construction". In: (1980). Ed. by Jonathan P. Seldin and J. Roger Hindley, p. 12. URL: http://lecomte.al.free.fr/ressources/PARIS8_LSL/Howard80.pdf.

[HP91] Robert Harper and Robert Pollack. "Type Checking with Universes". In: *Theor. Comput. Sci.* 89.1 (1991), pp. 107–136. DOI: 10.1016/0304-3975(90)90108-T. URL: https://doi.org/10.1016/0304-3975(90)90108-T.

[Hug95] John Hughes. "The Design of a Pretty-printing Library". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*. 1995, pp. 53–96. DOI: 10.1007/3-540-59451-5_3.

[Hur10] Chung-Kil Hur. "Heq: A Coq library for heterogeneous equality". In: *2nd Coq Workshop*. Edinburgh, United Kingdom, July 2010.

[Hut99] Graham Hutton. "A Tutorial on the Universality and Expressiveness of Fold". In: *J. Funct. Program.* 9.4 (1999), pp. 355–372. URL: http://journals.cambridge.org/action/displayAbstract?aid=44275.

[HWG13] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. "Unifying structured recursion schemes". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 2013, pp. 209–220. DOI: 10.1145/2500365.2500578.

[IG96] Ross Ihaka and Robert Gentleman. "R: A Language for Data Analysis and Graphics". In: *Journal of Computational and Graphical Statistics* 5.3 (1996), pp. 299–314. DOI: 10.1080/10618600.1996.10474713.

[Kah87] Gilles Kahn. "Natural Semantics". In: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. 1987, pp. 22–39. DOI: 10.1007/BFb0039592.

[Kel95] Richard A. Kelsey. "A Correspondence Between Continuation Passing Style and Static Single Assignment Form". In: *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. IR '95. San Francisco, California, USA: ACM, 1995, pp. 13–22. ISBN: 0-89791-754-5. DOI: 10.1145/202529.202532.

[Ken07] Andrew Kennedy. "Compiling with continuations, continued". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*. 2007, pp. 177–190. DOI: 10.1145/1291151.1291179.

[KG13] Hsiang-Shang Ko and Jeremy Gibbons. "Relational algebraic ornaments". In: *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*. 2013, pp. 37–48. DOI: 10.1145/2502409.2502413.

[KJ12] Steven Keuchel and Johan Jeuring. "Generic conversions of abstract syntax representations". In: *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2012, Copenhagen, Denmark, September 9-15, 2012*. 2012, pp. 57–68. DOI: 10.1145/2364394.2364403.

[Kle+09] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. "Formal C Semantics: CompCert and the C Standard". In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. 2014, pp. 543–548. DOI: 10.1007/978-3-319-08970-6_36.

[Kme15] Edward Kmett. *Bound*. Tech. rep. Dec. 2015. URL: https://www.schoolofhaskell.com/user/edwardk/bound.

[KP07] Andrew Kennedy and Benjamin C. Pierce. "On Decidability of Nominal Subtyping with Variance". In: *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*. Jan. 2007.

[LA04] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665.

[LB04] Fredrik Lindblad and Marcin Benke. "A Tool for Automated Theorem Proving in Agda". In: *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*. 2004, pp. 154–169. DOI: 10.1007/11617990_10.

[Ler09] Xavier Leroy. "Programming with dependent types: passing fad or useful tool?" IFIP Working Group 2.8 - Functional Programming. June 2009. URL: http://www.cs.ox.ac.uk/ralf.hinze/WG2.8/26/slides/xavier.pdf.

[Ler95] Xavier Leroy. "The ZINC experiment: an economical implementation of the ML language". In: (Nov. 1995).

[LH09] Daniel R. Licata and Robert Harper. "A universe of binding and computation". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. 2009, pp. 123–134. DOI: 10.1145/1596550.1596571.

[LM13] Sam Lindley and Conor McBride. "Hasochism: the pleasure and pain of dependently typed haskell programming". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. 2013, pp. 81–92. DOI: 10.1145/2503778.2503786.

[LMS10] Andres Löh, Conor McBride, and Wouter Swierstra. "A Tutorial Implementation of a Dependently Typed Lambda Calculus". In: *Fundam. Inform.* 102.2 (2010), pp. 177–207. DOI: 10.3233/FI-2010-304.

[Lut03] Marko Luther. "Elaboration and erasure in type theory". PhD thesis. University of Ulm, Germany, 2003. URL: http://d-nb.info/969720114.

[MAG07] Peter Morris, Thorsten Altenkirch, and Neil Ghani. "Constructing Strictly Positive Families". In: *Theory of Computing 2007. Proceedings of the Thirteenth Computing: The Australasian Theory Symposium (CATS2007). January 30 - Febuary 2, 2007, Ballarat, Victoria, Australia, Proceedings*. 2007, pp. 111–121. URL: http://crpit.com/abstracts/CRPITV65Morris.html.

[Mai18] The FreeBSD Project Maintainers. *FreeBSD Library Functions Manual*. Tech. rep. Feb. 2018. URL: https://www.freebsd.org/cgi/man.cgi?query=queue.

[mai18] The Ethereum project maintainers. *Solidity Language*. Tech. rep. Mar. 2018. URL: https://solidity.readthedocs.io/en/v0.4.20/.

[McB00] Conor McBride. "Elimination with a Motive". In: *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*. 2000, pp. 197–216. DOI: 10.1007/3-540-45842-5_13. URL: https://doi.org/10.1007/3-540-45842-5_13.

[McB04] Conor McBride. *Epigram*. Tech. rep. 2004. URL: http://www.dur.ac.uk/CARG/epigram.

[McB10a] Conor McBride. "Ornamental Algebras, Algebraic Ornaments". preprint on webpage at personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/Ornament.pdf. Aug. 2010.

[McB10b] Conor McBride. "Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation". In: *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010*. 2010, pp. 1–12. DOI: 10.1145/1863495.1863497.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire". In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. 1991, pp. 124–144. DOI: 10.1007/3540543961_7.

[MH10] Stefan Monnier and David Haguenauer. "Singleton types here, singleton types there, singleton types everywhere". In: *Proceedings of the 4th ACM Workshop Programming Languages meets Program Verification, PLPV 2010, Madrid, Spain, January 19, 2010*. 2010, pp. 1–8. DOI: 10.1145/1707790.1707792.

[MJ93] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*. 1993, pp. 259–270. URL: https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet.

[MKJ09] Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. "Algebra of programming in Agda: Dependent types for relational program derivation". In: *J. Funct. Program.* 19.5 (2009), pp. 545–579. DOI: 10.1017/S0956796809007345.

[MM04a] Conor McBride and James McKinna. "Functional pearl: i am not a number-i am a free variable". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*. 2004, pp. 1–9. DOI: 10.1145/1017472.1017477.

[MM04b] Conor McBride and James McKinna. "The view from the left". In: *J. Funct. Program.* 14.1 (2004), pp. 69–111. DOI: 10.1017/S0956796803004829.

[Mor+98] J. Gregory Morrisett et al. "From System F to Typed Assembly Language". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 1998, pp. 85–97. DOI: 10.1145/268946.268954.

[Mor06] Greg Morrisett. *F-Omega – the workhorse of modern compilers.* 2006. URL: https://web.archive.org/web/20160714072900/http://www.eecs.harvard.edu/~greg/cs256sp2005/lec16.txt (visited on 07/14/2016).

[Mou+15] Leonardo Mendonça de Moura et al. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26.

[MS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Napoli, 1984.

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007. ISBN: 978-1-84628-691-9. DOI: 10.1007/978-1-84628-692-6.

[Nor07] Ulf Norell. "Towards a practical programming language based on dependent type theory." PhD thesis. Chalmers University of Technology, 2007.

[Nor08] Ulf Norell. "Dependently Typed Programming in Agda". In: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. 2008, pp. 230–266. DOI: 10.1007/978-3-642-04652-0_5.

[PE88] Frank Pfenning and Conal Elliott. "Higher-Order Abstract Syntax". In: *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. 1988, pp. 199–208. DOI: 10.1145/53990.54010.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[Plo04a] Gordon D. Plotkin. "A structural approach to operational semantics". In: *J. Log. Algebr. Program.* 60-61 (2004), pp. 17–139.

[Plo04b] Gordon D. Plotkin. "The origins of structural operational semantics". In: *J. Log. Algebr. Program.* 60-61 (2004), pp. 3–15. DOI: 10.1016/j.jlap.2004.03.009.

[Plo75] G.D. Plotkin. "Call-by-name, call-by-value and the -calculus". In: *Theoretical Computer Science* 1.2 (1975), pp. 125–159. ISSN: 0304-3975. DOI: 10.1016/0304-3975(75)90017-1. URL: http://www.sciencedirect.com/science/article/pii/0304397575900171.

[PS99] Frank Pfenning and Carsten Schürmann. "System Description: Twelf - A Meta-Logical Framework for Deductive Systems". In: *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*. 1999, pp. 202–206. DOI: 10.1007/3-540-48660-7_14.

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[Rey03] John C. Reynolds. "What do types mean? — From intrinsic to extrinsic semantics". In: *Programming Methodology*. Ed. by Annabelle McIver and Carroll Morgan. New York, NY: Springer New York, 2003, pp. 309–327. ISBN: 978-0-387-21798-7. DOI: 10.1007/978-0-387-21798-7_15.

[Rey74] John C. Reynolds. "Towards a Theory of Type Structure". In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. London, UK, UK: Springer-Verlag, 1974, pp. 408–423. ISBN: 3-540-06859-7. URL: http://dl.acm.org/citation.cfm?id=647323.721503.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. "Liquid types". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implemen-*

tation, Tucson, AZ, USA, June 7-13, 2008. 2008, pp. 159–169. DOI: 10.1145/1375581.1375602.

[SA15] Josef Svenningsson and Emil Axelsson. "Combining deep and shallow embedding of domain-specific languages". In: *Computer Languages, Systems & Structures* 44 (2015), pp. 143–165. DOI: 10.1016/j.cl.2015.07.003.

[Sar97] Vijay Saraswat. *Java is Not Type-Safe*. Tech. rep. available at https://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html. AT&T Research, Aug. 1997.

[Sat12] Kazunori Sato. *An Inside Look at Google BigQuery*. Tech. rep. on webpage at https://cloud.google.com/files/BigQueryTechnicalWP.pdf. Google, 2012.

[SB17] Franck Slama and Edwin Brady. "Automatically Proving Equivalence by Type-Safe Reflection". In: *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*. 2017, pp. 40–55. DOI: 10.1007/978-3-319-62075-6_4.

[SC12] Bruno C. d. S. Oliveira and William R. Cook. "Functional programming with structured graphs". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. 2012, pp. 77–88. DOI: 10.1145/2364527.2364541.

[Sed88] Robert Sedgewick. *Algorithms, 2nd Edition*. Addison-Wesley, 1988, pp. 51–53. ISBN: 0-201-06673-4.

[Sew+07] Peter Sewell et al. "Ott: effective tool support for the working semanticist". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*. 2007, pp. 1–12. DOI: 10.1145/1291151.1291155.

[SJ02] Tim Sheard and Simon L. Peyton Jones. "Template metaprogramming for Haskell". In: *SIGPLAN Notices* 37.12 (2002), pp. 60–75. DOI: 10.1145/636517.636528.

[SL07] Tim Sheard and Nathan Linger. "Programming in Omega". In: *Central European Functional Programming School, Second Summer School, CEFP 2007, Cluj-Napoca, Romania, June 23-30, 2007, Revised Selected Lectures*. 2007, pp. 158–227. DOI: 10.1007/978-3-540-88059-2_5.

[Soz10] Matthieu Sozeau. "Equations: A Dependent Pattern-Matching Compiler". In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. 2010, pp. 419–434. DOI: 10.1007/978-3-642-14052-5_29.

[Str+12] Pierre-Yves Strub et al. "Self-Certification: Bootstrapping Certified Typecheckers in F* with Coq". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2012, pp. 571–584. URL: https://hal.inria.fr/inria-00628775.

[SV06] Alexandra Silva and Joost Visser. "Strong types for relational databases". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2006, Portland, Oregon, USA, September 17, 2006*. 2006, pp. 25–36. DOI: 10.1145/1159842.1159846.

[Swi12] Wouter Swierstra. "From Mathematics to Abstract Machine: A formal derivation of an executable Krivine machine". In: *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012*. 2012, pp. 163–177. DOI: 10.4204/EPTCS.76.10.

[TB15] Matúš Tejiščák and Edwin Brady. "Practical Erasure in Dependently Typed Languages". preprint on webpage at eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf. Feb. 2015.

[Tea18] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.7.2. Feb. 2018. URL: https://coq.inria.fr/distrib/current/files/Reference-Manual.pdf.

[UVP01] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. "Recursion Schemes from Comonads". In: *Nord. J. Comput.* 8.3 (2001), pp. 366–390. URL: http://www.cs.helsinki.fi/njc/References/uustaluvp2001:366.html.

[VW10] Dimitrios Vytiniotis and Stephanie Weirich. "Parametricity, type equality, and higher-order polymorphism". In: *J. Funct. Program.* 20.2 (2010), pp. 175–210. DOI: 10.1017/S0956796810000079.

[Wad01] Philip Wadler. "The Girard–Reynolds isomorphism". In: *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*. TACS '01. London, UK: Springer-Verlag, 2001, pp. 468–491.

[Wad89] Philip Wadler. "Theorems for Free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. 1989, pp. 347–359. DOI: 10.1145/99370.99404.

[Wad98] Philip Wadler. "A Prettier Printer". In: *Journal of Functional Programming*. Palgrave Macmillan, 1998, pp. 223–244.

[Wal12] Paul van der Walt. "Reflection in Agda". MA thesis. Department of Computing Science, Universiteit Utrecht, Oct. 2012.

[WB89] Philip Wadler and Stephen Blott. "How to Make ad-hoc Polymorphism Less ad-hoc". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. 1989, pp. 60–76. DOI: 10.1145/75277.75283.

[Wei02] Stephanie Weirich. "CIS 500: Software Foundations". Nov. 2002. URL: https://www.seas.upenn.edu/~cis500/cis500-f02/lectures/1106-4up.pdf.

[Wen+17a] Makarius Wenzel et al. *The Isabelle/Isar Implementation*. Oct. 2017, p. 159. URL: http://isabelle.in.tum.de/doc/implementation.pdf.

[Wen+17b] Makarius Wenzel et al. *The Isabelle/Isar Reference Manual,* Oct. 2017, pp. 158–166. URL: http://isabelle.in.tum.de/doc/isar-ref.pdf.

[WF94] Andrew K. Wright and Matthias Felleisen. "A Syntactic Approach to Type Soundness". In: *Inf. Comput.* 115.1 (1994), pp. 38–94. DOI: 10.1006/inco.1994.1093.

[WHE13] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. "System FC with explicit kind equality". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 2013, pp. 275–286. DOI: 10.1145/2500365.2500599.

[Wil+08] Reinhard Wilhelm et al. "The worst-case execution-time problem - overview of methods and survey of tools". In: *ACM Trans. Embedded Comput. Syst.* 7.3 (2008), 36:1–36:53. DOI: 10.1145/1347375.1347389.

[Woo18] Gavin Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. Tech. rep. on webpage at gavwood.com/paper.pdf. Mar. 2018.

[WS12] Paul van der Walt and Wouter Swierstra. "Engineering Proof by Reflection in Agda". In: *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*. 2012, pp. 157–173. DOI: 10.1007/978-3-642-41582-1_10.

[WW03] Geoffrey Washburn and Stephanie Weirich. "Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*. 2003, pp. 249–262. DOI: 10.1145/944705.944728.

[WYS11] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. "Binders unbound". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 2011, pp. 333–345. DOI: 10.1145/2034773.2034818.

[Xi07] Hongwei Xi. "Dependent ML An approach to practical programming with dependent types". In: *J. Funct. Program.* 17.2 (2007), pp. 215–286. DOI: 10.1017/S0956796806006216.

[Xi17] Hongwei Xi. "Applied Type System: An Approach to Practical Programming with Theorem-Proving". In: *CoRR* abs/1703.08683 (2017). arXiv: 1703.08683. URL: http://arxiv.org/abs/1703.08683.

[Xi99] Hongwei Xi. "Dead Code Elimination through Dependent Types". In: *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*. 1999, pp. 228–242. DOI: 10.1007/3-540-49201-1_16.

[XP98] Hongwei Xi and Frank Pfenning. "Eliminating Array Bound Checking Through Dependent Types". In: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*. 1998, pp. 249–257. DOI: 10.1145/277650.277732.